An Image Compression System for LEO Satellites

Eduard Kriegler

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Electronic Engineering at the University of Stellenbosch.

Study leader: Prof. S Mostert

December, 2003

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my original work, and that I have not previously, in its entirety or in part, submitted it at any university for a degree.

25 November 2003

Date

Abstract

Data volumes produced by the next generation of earth observation sensors have increased greatly in recent years. Sensors are generating more data than can be easily stored onboard satellites and transmitted to the ground-stations.

There are two strategies for solving this problem. The first is to process all images onboard the satellite, and only extract the useful or valuable information. The second is to use a compression algorithm to reduce the data volume.

This thesis looks at both strategies and then focusses on an evaluation of the Embedded Zerotree Wavelet (EZW) algorithm, a wavelet-based lossy image compression algorithm, as a solution to reduce the data volumes. Possible hardware implementation strategies for this algorithm are also explored. Finally, a suggested implementation of the EZW algorithm is compared with the FlexWave-II system and with JPEG2000.

Opsomming

Die data volumes wat deur die nuwe generasie van aardobservasiesensore geproduseer word, het dramaties vergroot in die laaste paar jaar. Daar word nou meer data geproduseer as wat aanboord van die satelliet gestoor kan word en meer as wat in die beperkte kommunikasietyd aan die grondstasie gestuur kan word.

Daar is twee strategiee om hierdie probleem aan te spreek. Eerstens kan beelde aanboord die satelliet verwerk word om die belangrikste of waardevolste inligting uit te haal en die res van die data word dan geskrap. Die alternatief is om 'n beeldkompressie-algoritme te gebruik om die data te verminder.

Hierdie tesis ondersoek hierdie strategieë en fokus dan op 'n evaluasie van die "Embedded Zerotree Wavelet"-algoritme. Die EZW-algoritme is 'n verlieserige, golfie-gebaseerde beeldkompressie-algoritme. Moontlike hardeware-implementeringsopsies word ondersoek en die resultate van een voorgestelde opsie word vergelyk met die FlexWave-II stelsel asook die nuwe JPEG2000-standaard.

Acknowledgements

I would like to thank the following people:

- My supervisor, Prof Sias Mostert, for his encouragement and guidance
- Mow-Song, Ng for all his assistance in getting the EZW algorithm working
- Miss L. Mitford-Barberton for her support and proof reading this document
- Francois Retief for his all advice especially with the Cygwin environment and with the GNU C-Tools
- All my other friends and colleagues for their support

Contents

List of Abbreviations and Acronyms

ASIC	-	Application Specific Integrated Circuit
bpp	-	Bits Per Pixel
bps	-	Bits per second
CAS	-	Column Address Strobe
dB	-	Decibels
DCT	-	Discrete Cosine Transform
DDR	-	Double Data Rate
DFT	-	Discrete Fourier Transform
DLL	-	Delay-Locked Loop
DN	-	Digital Number
DSP	-	Digital Signal Processor
DWT	-	Discrete Wavelet Transform
ESA	-	European Space Agency
EZW	-	Embedded Zerotree Wavelet
FFT	-	Fast Fourier Transform
FIFO	-	First In First Out
FPGA	-	Field Programmable Gate Array
GCP	-	Ground Control Point
GHz, MHz	-	Gigahertz, Megahertz
HDL	-	Hardware Description Language
IC	-	Integrated Circuit
IEC	-	International Electrotechnical Commission
IEEE	-	Institute of Electrical & Electronics Engineers
IO	-	Input Output
IR	-	Infrared
ISO	-	International Organisation for Standardisation
JPEG	-	Joint Picture Experts Group
kB, MB	-	Kilobyte, Megabyte
KLT	-	Kahrunen-Loewe Transform

LEO	-	Low Earth Orbit
MSE	-	Mean Square Error
MSMI	-	Micro-Satellite Multi-sensor Imager
MUX	-	Multiplexor
NDVI	-	Normalised Difference Vegetation Index
OBC	-	Onboard Computer
PC	-	Personal Computer
PCA	-	Principle Component Analysis
PLL	-	Phase Lock Loop
PRNU	-	Photo-Response Non-Uniformity
PSNR	-	Peak Signal to Noise Ratio
RAM	-	Random Access Memory
SDRAM	-	Synchronous Dynamic Random Access Memory
SPIHT	-	Set Partitioning in Hierarchical Trees
SRAM	-	Static Random Access Memory
TC	-	Tasselled Cap
VHDL	-	Very high speed integrated circuit Hardware Description Language
VNIR	-	Very Near Infrared
WCET	-	Worst-Case Execution Time

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the last few years, many applications which utilise the images taken by earth observation systems have been demanding more detailed data. The result has been an increase in the amount of data produced by new earth observation sensors.

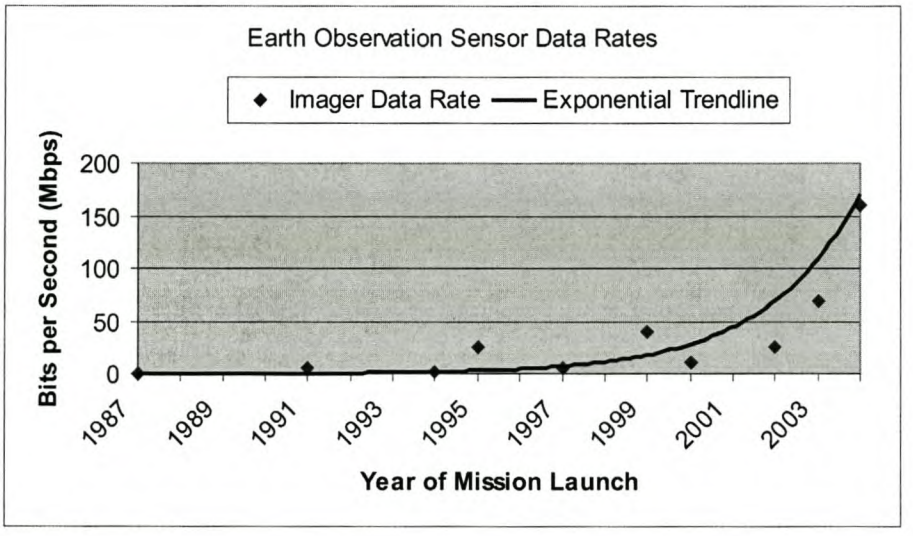

Figure 1.1: *Tendency in Satellite Imager Data Rates*

Figure 1.1 illustrates the recent exponential increase in the data output rates of satellite imagers. The satellite imagers included in the graph are listed in Table 1.1. Later imagers record more spectral bands with more pixels per line, and thus result in higher data output rates. For example, the next generation of earth observation sensors for the SunSpace MSMI (Micro-Satellite Multi-sensor Imager) programme will generate more than 1000 Mbps.

One problem caused by the high data rates, is data storage onboard satellites. Data is

Satellite and/or Imager	Year of Launch	Data Rate
SSM/I	1988	4.66 kbps
IRS1b - LISS-I	1991	5.2 Mbps
GOES I-Q	1994	2.6 Mbps
SPOT-HRVIR	1995	25 Mbps
CBERS-1 - IRMSS	1997	6.13 Mbps
SUNSAT	1999	40 Mbps
MODIS	2000	10.6 Mbps
Envisat - MRISI	2002	25 Mbps
IRS P5 - LISS-IV	2003	70 Mbps
ALOS - AVNIR-2	2004	160 Mbps

Table 1.1: *Satellite Imager Data Rates*

produced while the satellite is not within communication range of its ground-station. Very large memory systems are therefore required to store the data until it can be transmitted to the ground-station. One example is LandSat 7 which has an onboard solid state recorder with almost 50 gigabyte capacity.

Most images produced by satellite imagers are used in specific scientific applications. Only some of the information contained in the images is required by the applications. Currently, as demonstrated by the 'Current Process' illustrated in Figure 1.2, the required information is extracted from the images after the images had been downloaded to the ground-station.

The first solution, Solution A, situates the information extraction system onboard the satellite. Only the results of the information extraction are transmitted to the ground-station, where it is added to the database. However, Solution A requires human input and *in situ* data (for instance, the atmospheric conditions prevailing at the time the image was taken) to be effective, and is therefore not feasible.

Solution B utilises an image compression system onboard the satellite to reduce the data volume. The information extraction is, as in the current process, performed on the ground.

MSMI sensors generate more than 1000 Mbps. X-Band data transmitters are able to transmit at a data rate of about 150 Mbps. LEO satellites are, on average, in communication range with their ground-stations for about 30 minutes per day. For such satellites, to download all the data generated during a 24 hour period will require a compression ratio of 320:1. In order to reduce this ratio, the satellite may carry more than one transmitter, or there may be more than one ground-station. Another possibility is that the imager

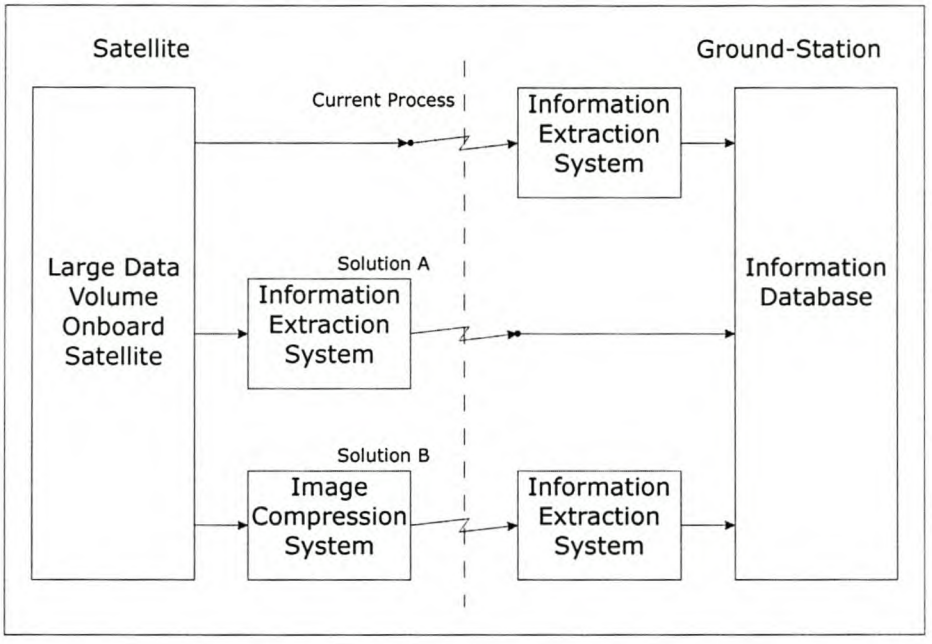

Figure 1.2: *Two Solutions to the Data Volume Problem*

may not constantly operate (due to cloud cover, bad lighting or simply because there is nothing interesting in view), and therefore the compression ratio need not be as high as 320:1. A ratio of at least 20:1 will be required. At a ratio of 20:1, the imager can operate for 1.5 hours per day when using one downlink channel and one ground-station.

Image compression algorithms can be divided in two main groups: lossless and lossy. Lossless algorithms guarantee perfect reconstruction of the original data, but the compression ratio is limited by the entropy of the original data. Natural images typically cannot be compressed more than 2:1 with lossless algorithms.

Lossy compression algorithms offer much higher compression ratios than lossless algorithms, but the trade off is that some data is lost. Perfect reconstruction is not possible with lossy compression algorithms.

The most commonly used lossy image compression algorithm is the DCT-based JPEG algorithm. The DCT-based JPEG algorithm is known for the blocking-effect which is visible at high compression ratios. The new wavelet-based compression algorithms have solved the blocking-effect, and are able to produce higher quality results at high compression ratios. A joint project between IMEC and ESA has produced the FlexWave-II system which uses a wavelet transform and a zerotree encoder to compress images.

This thesis will investigate the two solutions mentioned above. An understanding of the image processing chain used in Solution A may lead to methods of improving Solution B, thereby improving the results obtained by lossy compression. The influence of lossy

image compression on image corrections and information extraction is also discussed. Then the focus shifts to the design of a wavelet-based image compression system for micro-satellites. An important consideration of the design process is to ensure that the system can run, in real-time, in a satellite environment.

1.2 Document Overview

- Chapter 1: **Introduction**
  This chapter provides background information on the thesis subject.

- Chapter 2: **Image Processing and Information Extraction**
  The processing chain for extracting useful information from satellite images is explored. The influence that image corrections may have on lossy compression is also investigated.

- Chapter 3: **Image Compression System**
  The structure of an image compression system as well as the considerations for choosing the algorithms are discussed in this chapter. The EZW algorithm and the wavelet transform are introduced.

- Chapter 4: **Wavelets and Embedded Zerotree Wavelet Encoding**
  The concept of wavelets, the wavelet transform and the Embedded Zerotree Wavelet algorithm are explained in detail.

- Chapter 5: **Implementation Options**
  This chapter covers the different hardware implementation options that were evaluated during the research.

- Chapter 6: **Wavelet Implementation**
  The details of a VHDL implementation of the wavelet transform are given.

- Chapter 7: **EZW Implementation on an Embedded Processor**
  The process of selecting an embedded processor, and a performance analysis of the implementation, is covered in detail.

- Chapter 8: **Comparisons with Commercial Systems**
  The performance of the system is compared to that of IMEC and ESA's FlexWave-II system, and also Amphion's CS6510 JPEG2000 encoder.

- Chapter 9: **Conclusions and Recommendations**
  This chapter summarises the advantages and disadvantages of the image compression system, and makes some recommendations for improvement on the current design.

Chapter 2

Image Processing and Information Extraction

This chapter will examine the feasibility of the first solution, Solution A, which was discussed in Chapter 1.

Before any information extraction can take place a series of image corrections must be completed. Section 2.1 investigates the processing steps that constitute the information extraction system. Even if the information extraction is not performed onboard the satellite, some of the image correction steps may improve the lossy compression. It is also possible that the lossy compression may have an adverse effect on performing the image corrections at a later time. The influences that image corrections and lossy compression may have on each other are discussed in Section 2.2.

2.1 The Processing Chain

The processing chain consists of three image corrections: system, atmospheric and geometric. After the corrections have been made, images are resampled in a new coordinate system, for instance, in real earth coordinates. Finally, information extraction routines are run on the image.

2.1.1 System Corrections

Both area and linear imaging sensors have fixed pattern noise. One component of the noise is independent of the illumination level of the pixel, and is related to dark current differences and other offsets between pixels. A second component is proportional to signal level (PRNU - photo-response non-uniformity), and results from variation in the photo-response of pixels arising from small geometric differences from masks, lithography, and

etching.

The first image correction is thus a per-pixel offset and gain correction. The offset correction also normally removes any clock feed-through artefacts from the data.

On area arrays, it is possible to develop line dropouts due to failure of one of the transfer cells. To avoid the effects of such a line being spread further by a compression algorithm, the line should be replaced by the nearest valid line prior to compression.

Some imaging sensors have an analogue output. For these sensors, the above corrections can be done with analogue electronics. For sensors with digital outputs, a digital signal processor (DSP) will have to multiply each pixel by a gain-constant and add an offset to it. A small, dedicated processor would be sufficient for the task.

2.1.2 Atmospheric Corrections

The next step after system corrections is to correct for the effect of the atmosphere. The atmosphere, and particles contained in the atmosphere, reflect, refract and absorb electromagnetic waves (See Figure 2.1). The resulting effective radiation seen by the sensor includes part of the radiation/reflection of the target, due to scattering and absorption, as well as false radiation caused by scattering within the atmosphere. These effects are not uniform, and vary as temperature, pressure, particle distribution and distance to the target varies. Effective correction of these effects requires a detailed model of the atmosphere and some knowledge of the atmospheric condition at the time of image acquisition.

One good model is the 6S-model [16]. As input it requires information on the geometrical conditions, the time and date, the latitude and longitude, the atmospheric model (ie. tropical, desert, etc.), the aerosol model, the aerosol concentration, altitude of the target and sensor, and on which frequency-band the image is. A satellite typically will not know which atmospheric model or aerosol model and concentration to use, and therefore human intervention may be required to effectively use the 6S-model.

2.1.3 Geometric Correction and Referencing

Geometric correction removes the effects of optic viewing geometry, satellite orientation, land slope and shadows in images, while geometric referencing is the process of mapping the pixels of an image to actual earth coordinates. The level of processing that is appropriate depends to a large extent on the purpose to which the images will be put.

The new technique of using ground control points (GCPs) to warp and resample images can perform both the geometric correction and geometric referencing in one step, provided that a sufficient number of GCPs have been identified in the image.

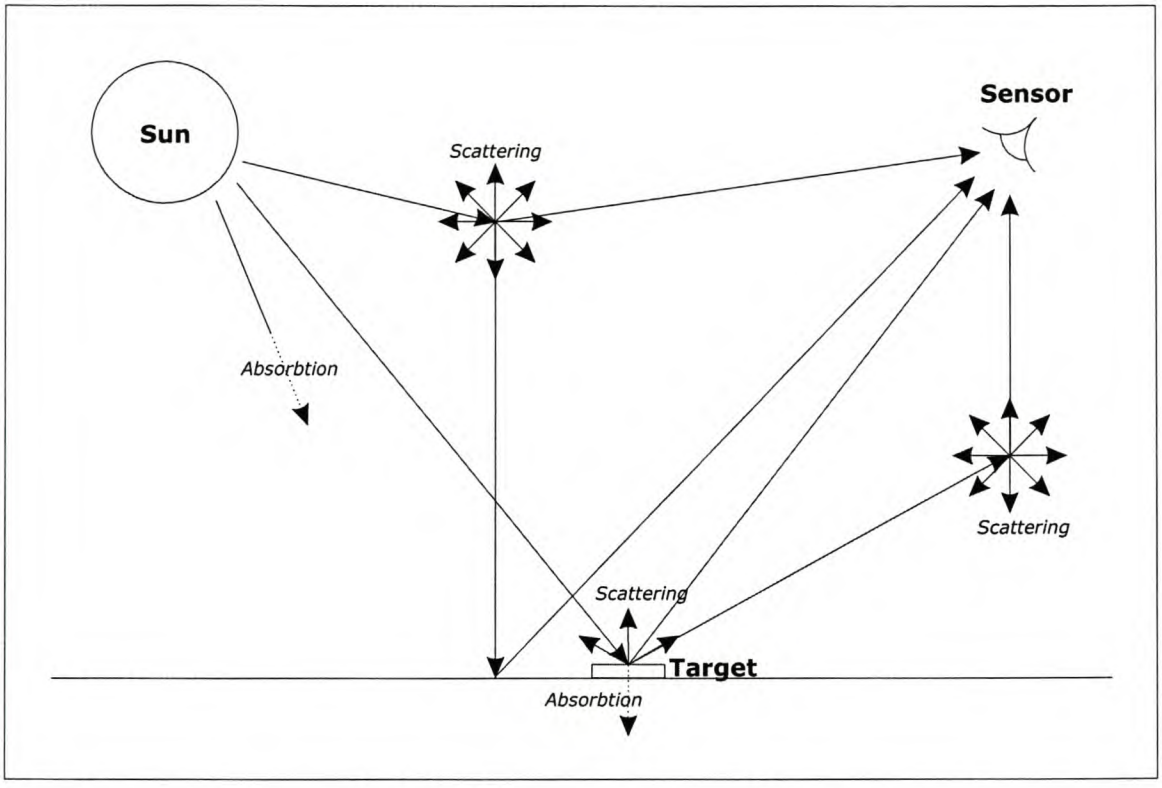

Figure 2.1: *Effects caused by the Atmosphere*

The GCP-process works as follows. Firstly, GCPs (for instance a characteristic bend in a river, a large cliff or large man-made objects like bridges) are identified. Polynomial equations are then generated which will map the GCP pixels in the images to the actual coordinates of the real object. These equations are then used to warp the whole image, after which it is resampled in the new coordinate system. The geometric correction/referencing is more successful the greater the number of GCPs used.

Currently, GCPs are identified by studying satellite images and then going out into the field to identify objects visible in the images. This process may be automated in satellites if a large database of GCPs can be created. Automating the process would result in image recognition algorithms being used to find the GCP-objects in satellite images. Such an automation would be a big undertaking, and more research on the subject is required.

2.1.4 Information Extraction

Once all the appropriate corrections have been completed, the information extraction process can start. Some examples of methods used to extract information include the creation of ratio-images, transformations such as the principal component analysis, and digital image classification.

2.1.4.1 Creation of Ratio-Images

A ratio-image is created by taking ratios of two images of different spectral bands. These ratios can be used to identify different mineral content, vegetation and urban areas (amongst many others). One common ratio-image is the Normalised Difference Vegetation Index (NDVI). The formula for the NDVI is given by Equation 2.1.

$$\text{NDVI} = \frac{\text{VNIR} - \text{red}}{\text{VNIR} + \text{red}} \tag{2.1}$$

High values for the NDVI indicate dense vegetation. Values close to zero indicate bare soil and negative values indicate water. The NDVI is only a numerical measure of photosynthesis in the target area and do not give any information as to the type of plant growth.

Many other ratios are used. Over fifty vegetation indices exists. Each ratio provides information that is not available in a single band.

Although calculating these ratios is not a complex process (the algorithmic complexity is a linear function of the number of pixels comprising the image), many ratio-images are required to get a clear picture of what is happening in the target area. Ratio-images will not produce the required data volume reduction.

2.1.4.2 Linear Data Transformations

Linear data transformations are used to rearrange multidimensional data sets (such as multi-spectral imagery) in order to:

1. Decorrelate the information in the new dimensions (Principle Component Analysis or PCA)

2. Directly relate the information in the new dimensions to scene characteristics such as soil and vegetation (Tasselled Cap or TC transformation)

3. Reduce data dimensionality by concentrating relevant information into fewer dimensions (both PCA and TC)

4. Maximise the separability between predetermined feature classes while minimising the variability within the classes (canonical transformation)

PCA (also known as the Kahrunen-Loewe transform) is the most commonly used linear transformation, but it creates scene-dependent results. Therefore the PCA transformation should not be used in applications where a multi-temporal analysis is required. The PCA transformation is effective in reducing the number of bands that need to be stored. The

first three principle components of the multi-band LandSat and Spot data sets contain over 90% of the variance.

The tasselled cap transformation (also known as the Kauth-Thomas transformation) overcomes part of the scene-dependency of PCA. The tasselled cap approach transforms the data such that the soil, vegetation and moisture information are projected onto separate planes in a multidimensional data space.

Although both PCA and the tasselled cap transformation can reduce the data volume, this reduction is less than the minimum required reduction of 20:1.

2.1.4.3 Digital Image Classification

Digital image classification includes all procedures that map image pixels to information classes or feature categories.

The images produced by the imaging hardware contain the reflectance and radiation of targets on the ground at different wavelengths (green, red, IR, etc). The spectral response values held in pixels are also called the digital number (DN). These digital numbers of pixels are compared to libraries of spectral responses of different plants and soil types.

Figure 2.2 illustrates how pixels may be classified in a system where only two bands (or wavelengths) are used. Firstly, the library defines regions that may be considered as urban, forest, water, etc. Any pixel that falls inside one of these regions will be considered to represent that class. However, not all pixels will fall inside one of these regions. The unclassified pixel in the figure is not close to any of the regions. Pixels such as that one are usually classified according to its distance to the mean of each region.

The above method is an example of supervised classification. In unsupervised classification, there is no starting library. The classification software will create new classes from the pixel values in the images. Since the software cannot know what type of vegetation or soil each class represents, the classes created by unsupervised classification must be identified at a later stage.

The results of digital classification are not always very accurate. Therefore, after classification, the results are measured against reference images, enabling calculation of the measure of accuracy. One common measure is Cohen's coefficient of agreement. Cohen's coefficient of agreement indicates proportionally how much better the results are than a purely random class assignment. The classification is repeated with tweaked settings until a sufficiently high accuracy is achieved.

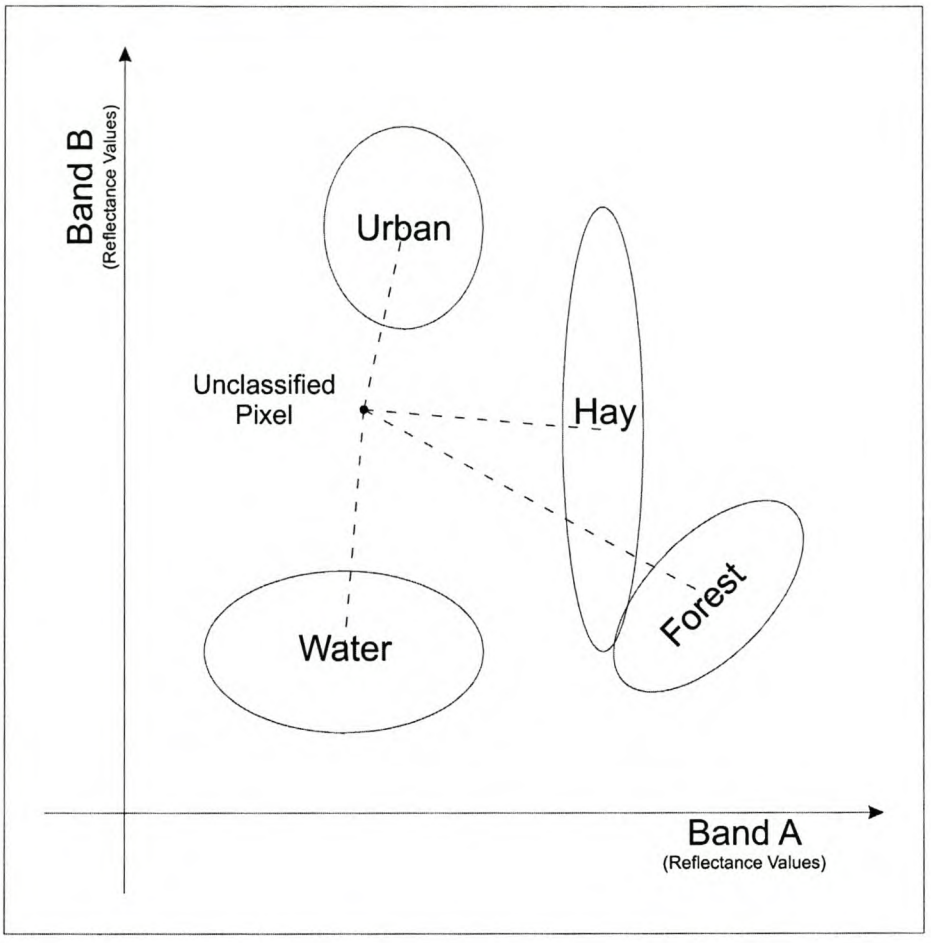

Figure 2.2: *Example of Digital Image Classification*

2.2 Influence of Image Corrections on Image Compression

If the image processing is to be done on the ground, a lossy image compression algorithm will be needed to obtain the significant data reduction required to get the data on the ground. If a lossy image compression algorithm is used, what effect will it have on the image correction if it is performed later? And conversely, how will the image correction affect the compression if it is performed before the compression? This section will investigate these two questions.

System corrections remove noise and line dropout artefacts. The image compression will add more noise (quantisation noise) to the image, and artefacts may tend to get spread out to neighbouring pixels due to lossy compression. Also, the additional noise in the images will tend to reduce the inter-pixel redundancies, which are exactly what the compression algorithm will exploit to obtain good compression results. The system noise

will compromise the effectiveness of the compression algorithm, which will then in turn compromise the effectiveness of the system corrections.

Atmospheric correction also removes some artefacts. These artefacts will also be spread out if they are not removed prior to compression. However, atmospheric correction generally deals more with correcting radiation intensity levels which are already spread across many neighbouring pixels. Lossy image compression tends to damage images in areas where there is a large contrast between neighbouring pixels. It can be concluded that the atmospheric correction and lossy compression will not influence each other noticeably.

Geometric correction may be more difficult to perform after lossy compression. Shadows and errant radiation levels caused by slopes are normally localised to a few neighbouring pixels and again their effects will get spread out.

Neither the artefacts caused by the atmosphere nor those caused by geometric distortions have a big influence on inter-pixel redundancies. Thus they will have a very limited influence on the effectiveness of the lossy compression.

From a data quality point of view, all the image corrections must be performed prior to information extraction to ensure the highest quality. If lossy image compression is performed at some point in the processing chain, the quality will suffer, since lossy image compression cannot guarantee data quality. To improve data quality, it is important to perform all processing steps that influence lossy compression prior to the compression itself. The system corrections have the largest influence on the effectiveness of the lossy image compression and should therefore be performed before compressing the images.

The next section will look at the effect that lossy image compression may have on the information extraction procedures.

2.3 Lossy Image Compression and Information Extraction

Lossy image compression means that, when the compressed data is uncompressed, the reconstructed image will not be an exact copy of the original. The image will still be the same size and all the features in the image should still be in the same locations, but the actual values of the pixels may differ.

Common problems encountered when using lossy image compression include the following:

- The blocking effect - The blocking effect is common with JPEG and other DCT-based compression algorithms. The reason for the blocking effect is that the image is divided into small blocks, which are then compressed separately. When the image

is reconstructed, the edges of neighbouring blocks do not always match, resulting in the blocking effect.

- Ringing - Edges in the image get blurred. This is caused by wavelet-based algorithms using excessively long wavelet filters or too large DCT blocks in DCT-based algorithms.
- Blurring - Blurring of the image occurs at very high compression ratios.
- DC leakage - The effect seen in images are a tartan pattern superimposed on the image. DC leakage occurs when using non-regular filters.
- Edge Effects - Edge effects are caused by false discontinuities at edges. When using symmetric wavelets, images can be reflected at the edges to remove the false discontinuity, thereby fixing the problem.

If pixel-errors are small compared to the actual pixel value, ratio-images can reject it. The same is true for decorrelation transformations like the PCA. Other types of transformations, for instance the TC transformation, will propagate the errors.

In the case of digital image classification, small errors can produce serious errors. Some pixels, even without the effect of lossy compression, are difficult to classify because they do not fall inside any classes. Some pixels lie on or near the border between two classes. Even if lossy compression causes only small pixel errors, the result may be that these pixels are classified in a different class than they would have been.

2.4 Conclusion

Performing the information extraction onboard satellites is not feasible at this time. The reasons for this conclusion are as follows:

1. Atmospheric correction requires knowledge of atmospheric conditions which will not always be available to the satellite when it must perform this correction.
2. The geometric correction and referencing process requires human intervention to deliver good results. More research on this subject will be required to automate it.
3. The information extraction processes explored either do not produce sufficient data volume reduction (Ratio-images and linear data transformations) or requires some amount of human control to achieve better results.

Even though the information extraction is not viable on the satellite, it is recommended that at least the system corrections be performed onboard the satellite prior to lossy image compression. Performing the system corrections will improve the image quality after the lossy image compression.

Chapter 3

The Image Compression System

There are two main groups of image compression systems: lossless and lossy. Lossless systems compress images in such a way that they can be perfectly reconstructed. Lossy compression systems produce the highest compression ratios, but this is at the cost of losing some image information.

The most common method for implementation of lossy image compression is in the use of a transform-based image compression system.

3.1 Inside a Transform-based Image Compression System

Transform-based image compression systems consist of three sections: the mapper (or transformer); the quantiser; and the entropy encoder (See Figure 3.1).

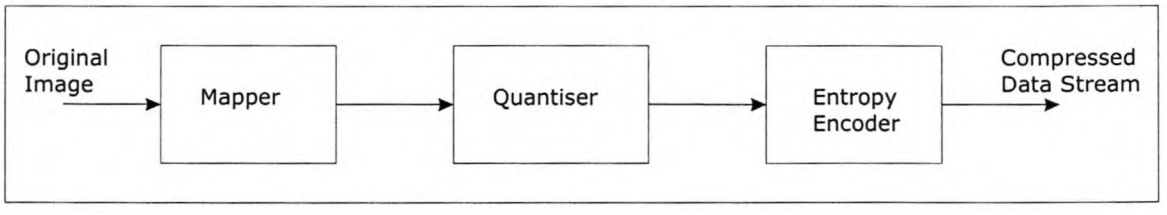

Figure 3.1: *A Typical Transform-Based Image Compression System*

The mapper transforms the image from the space/time domain to a new domain where the inter-pixel redundancies are reduced. Different mappers will transform the image to different domains. Some examples of commonly used mappers include the discrete fourier transform (DFT) or fast fourier transform (FFT), the discrete cosine transform (DCT), the Kahrunen-Loewe transform (KLT) and the discrete wavelet transform (DWT). Typically the mapper is a lossless or reversible process, but this is not always the case.

The aim of quantisation is to reduce the amount of data used to represent the information within the new domain. Quantisation is in most cases not a reversible operation. It therefore belongs to the class of so called 'lossy' methods.

The output of the quantiser is passed to an entropy encoder which reduces coding redundancies. It is possible to omit the entropy encoder, but doing this has an adverse effect on the compression ratio. Examples of entropy encoders are the Huffman encoder, arithmetic encoder and the RangeCoder [12]. Entropy encoders are lossless.

3.2 Measuring Algorithm Performance

A common way of measuring the performance of image compression algorithms is to compare image quality and the compression ratio. To compare different compression algorithms, plots of image quality versus compression ratio were made. Figure 3.2 shows a comparison between EZW, and DCT-based JPEG. The comparison was done with an eight bit per pixel grey-scale copy of the well-known Lena test-image.

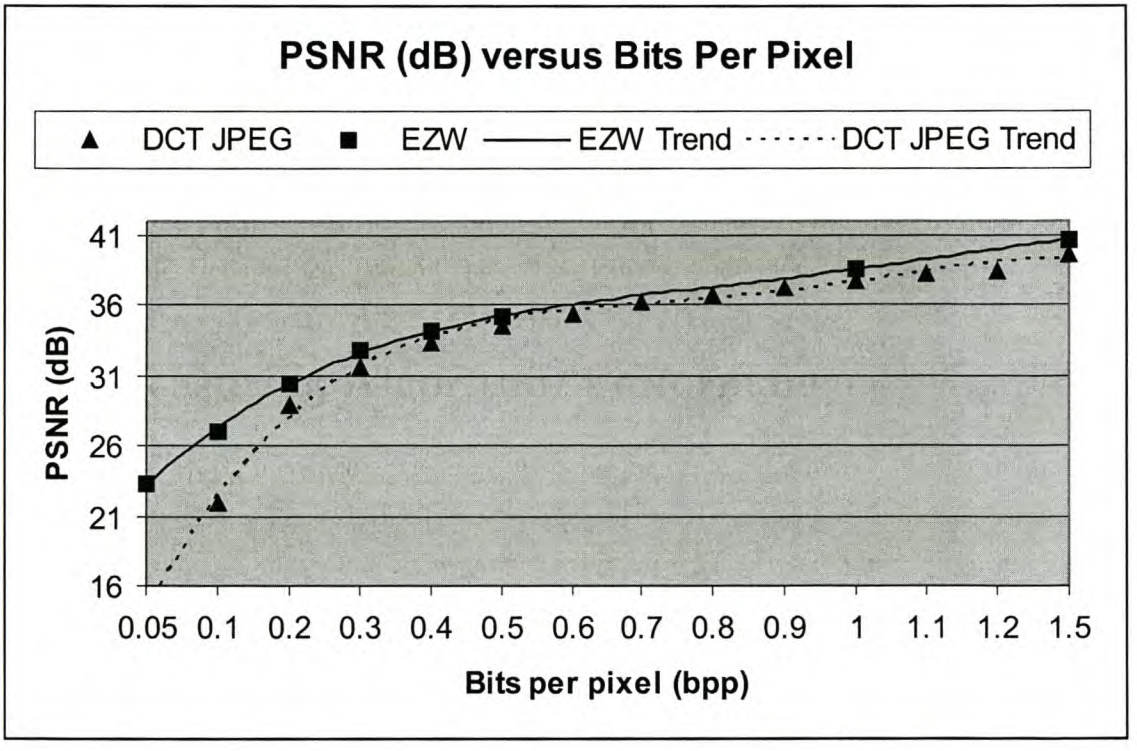

Figure 3.2: *Comparison of EZW and DCT-based JPEG*

The EZW algorithm outperforms DCT-based JPEG at low bit-rates (below 0.4 bpp or 20:1 compression ratio if the source data is eight bits wide).

The next two subsections will cover the details of measuring image quality and the compression ratio.

3.2.1 Measuring Image Quality

Measuring image quality depends on the definition of image quality. In some applications, for instance publishing digital photographs on the internet, the object is to have the image still look good to the eye. In this case, image quality would be defined as the quality perceived by the human eye. For scientific applications, on the other hand, the quality factor may be defined completely differently.

The peak-signal-to-noise-ratio (PSNR) is generally accepted as a good measure of quality as perceived by the human eye. PSNR is the most often-used measure of image quality, especially in image compression applications. PSNR is calculated as follows:

$$\text{PSNR} = 10 \log_{10} \left(\frac{(2^B - 1)^2}{\text{MSE}} \right) \tag{3.1}$$

where B is the bits per pixel of the original image. MSE is the mean square error. Equation 3.2 gives the formula for calculating the MSE.

$$\text{MSE} = \frac{1}{N_1 N_2} \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} (x[n_1, n_2] - \hat{x}[n_1, n_2])^2 \tag{3.2}$$

where the image dimensions are $N_1 \times N_2$ pixels; $x[n_1, n_2]$ is the pixels of the original image and $\hat{x}[n_1, n_2]$ the pixels of the reconstructed image.

For scientific applications the measure used is the absolute maximum pixel error and average absolute pixel error. These are obtained by calculating a per-pixel absolute difference image from the original and reconstructed images. These measures give a good indication as to the amount of data disruption caused by the compression algorithm, and hence the impact on pixel digital value numbers used for automatic classification algorithms.

3.2.2 Compression Ratio

The compression ratio is the ratio of the size of the compressed result versus the size of the original data. For general compression algorithms, compression ratio is commonly expressed as the percentage that the output size is of the input size. However, with image compression algorithms the norm is to express the compression ratio as bits per pixel (bpp). Bits per pixel is obtained by dividing the output size in bits by the number of pixels in the image.

3.3 Considerations for Choosing the Algorithms

The following constraints need to be considered when selecting the algorithms that incorporate the image compression system:

- The algorithm should compress images as near to real-time as possible. The MSMI imaging sensors produce data at a rate of 1000 Mbps. In order not to limit the duration data is recorded from these sensors, the image compression algorithms should be able to process images at the same rate.
- A compression ratio of at least 20:1 is required. This can be calculated from the output data rate of the imager subsystem, the data capacity of the datalink with the ground-station and the expected duty cycle of the imager subsystem. The compression algorithm should show favourable image quality results at that and higher ratios.
- Memory usage and computational complexity need to be minimised, as they are expensive in satellites in terms of physical space, power usage, and execution time. High memory usage requires more external memory and therefore more components, thus making the system larger. Computationally complex algorithms require higher speed processors that use more power than the more conventional processors.
- For operation in the space environment the radiation tolerance of the hardware must also be considered. Radiation upsets must not influence the operation of the system.

3.4 Wavelets and Embedded Zerotree Encoding

Due to the blocking-effect caused by the DCT-based image compression algorithms, it was decided to implement a wavelet-based algorithm. The two main algorithms that were considered were the Embedded Zerotree Wavelet (EZW) algorithm and the JPEG2000 algorithm. The EZW algorithm was selected because it is a simpler algorithm with fewer processing steps. The EZW algorithm only requires integer manipulation, and rarely uses multiplication. It never uses division, which can be expensive to implement.

Memory usage with the EZW algorithm is moderate. The algorithm requires memory to store the image being compressed, as well as to keep track of what has been encoded and what must still be encoded. The total memory usage in the final implementation was a little more than three times that required by the image, but further optimisation is possible.

In order for the EZW algorithm to be an effective quantiser, a wavelet transform must be used as the mapper. A very popular wavelet transform is the lifting algorithm which can be implemented using only integer arithmetic. This algorithm is covered in detail in Section 4.1.2.

Chapter 4

Wavelets and Embedded Zerotree Wavelet Encoding

This chapter will discuss wavelets, the wavelet transform and an algorithm for calculating the wavelet transform, as well as the EZW algorithm, in detail. The wavelet transform (the lifting algorithm in particular) and the EZW algorithm are the main components of the image compression system. A good understanding of both are necessary in order to make suitable design choices when implementing the algorithms.

4.1 Wavelets

4.1.1 What are Wavelets?

Wavelets are localised functions in time (or, for images, in space). They are used to create a wavelet basis which is in turn used to transform data from a time or space domain to a joint time-frequency domain. A wavelet basis is derived from the wavelet by dilations and translations.

One of the first wavelets created was the Haar-wavelet. Equation 4.1 shows the Haar function (more wavelets are shown in Section 4.1.3):

$$w(t) = \begin{cases} 1 & 0 \leq t < 0.5 \\ -1 & 0.5 \leq t < 1 \end{cases} \tag{4.1}$$

The Haar wavelet transform will compute the averages and differences of an input sequence of values. For example, given an input sequence of $\{a, b\}$, the result will be $s = (a+b)/2$ and $d = (b-a)$. The values of a and b can be reconstructed as $a = s - d/2$ and $b = s + d/2$. The s-result is an approximation of the original signal (or the lower frequency band of the original signal) and the d-result is detail information (or high frequency band) required to reproduce the original signal.

The wavelet transform is fully reversible without data loss. Data loss is incurred by the quantisation step. This can be illustrated by an example. Given a vector $[13, 17, 16, 15]$ the Haar-wavelet transform will be $s = [15, 15.5]$ and $d = [4, -1]$. If the quantiser simply rounds the wavelet coefficients to the nearest integer, the s-vector will become $s = [15, 16]$. If the original vector is reconstructed using this new s-vector, the result will be $[13, 17, 16.5, 15.5]$. This error introduced in the vector is called quantisation noise.

The wavelet transform illustrated in the previous paragraph is known as the discrete wavelet transform (DWT). The Discrete Wavelet Transform is a recurrent process of low- and high-pass filtering performed on the input data set (See Figure 4.1). In the first iteration, the whole data vector is low- and high-pass filtered, and the resulting vectors stored separately. These two vectors will each have the same length as the original vector. Since the expansion of the data is undesired, the two vectors are downsampled. This means that every second element in the vectors is removed. The low pass filtered result is the approximation of the original signal and the high pass filtered result the detail coefficients (also called wavelet coefficients). Together these two resulting vectors form a first scale wavelet transformation of the original signal.

The second and subsequent iterations of the filtering process are performed on the vector (array) generated by the previous iteration's low-pass filter. The vector (array) from the high-pass filter is stored. Each new iteration adds another scale to the wavelet-transformed image, as illustrated in Figure 4.1.

In order to transform images, a two-dimensional transform is needed. The Square Two-Dimensional DWT is calculated using a series of one dimensional DWTs:

- Step 1: Replace each image row with its 1D DWT
- Step 2: Replace each image column with its 1D DWT
- Step 3: Repeat steps (1) and (2) on the lowest subband to create the next scale
- Step 4: Repeat step (3) until the desired number of scales had been created

Figure 4.2 shows the well known Lena-image in different steps of the two dimensional DWT. The first image is the original non-transformed image. In the second image all the rows have been transformed once (this is the state after step 1 above). The third and fourth images show transformed images with one and two wavelet scales respectively. The wavelet scale refers to the number of times the transform was applied to the original image. In the fourth image it is clear that only the lowest subband was transformed during the second iteration.

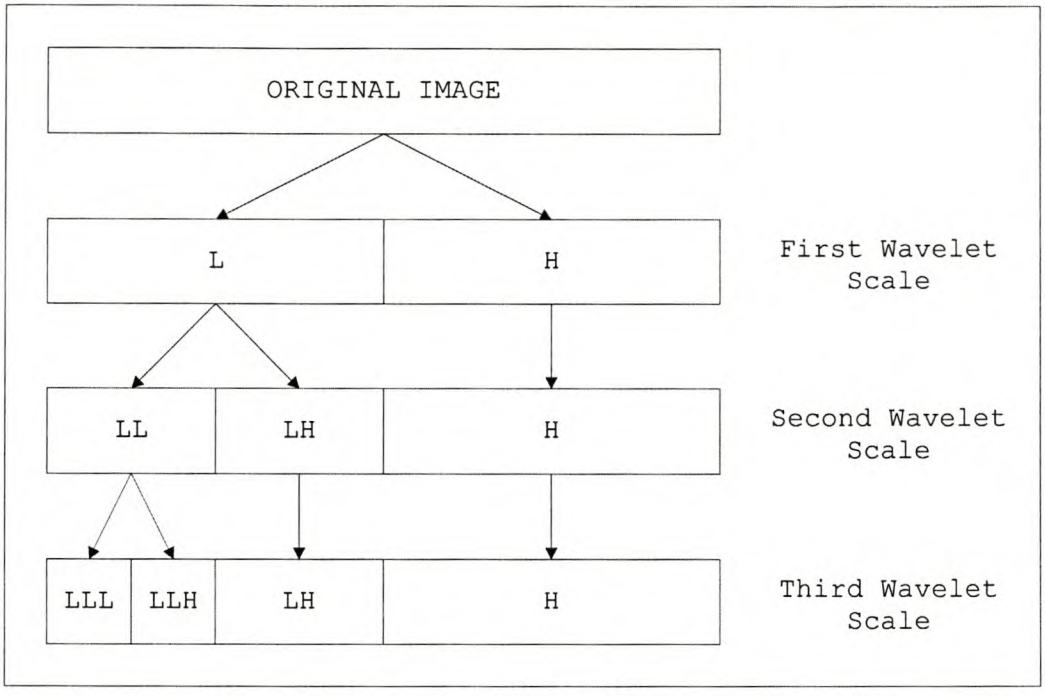

Figure 4.1: *One Dimensional DWT*

This process of filtering can be calculated with a convolution-based filtering algorithm, but the lifting algorithm is now the preferred (or most commonly used) algorithm for computing wavelet transforms. The reasons for this preference are [14]:

1. It allows a faster implementation of the wavelet transform. Asymptotically, for long filters, the cost of the lifting algorithm for computing the wavelet transform is half that of the convolution algorithm [6]. For the popular $(9, 7)$ biorthogonal wavelet filter, a speed up of 64% is reported.

2. Lifting allows an in-place calculation of the wavelet transform. The algorithm will replace the original signal with its transform. No extra or temporary memory is required.

4.1.2 The Lifting Algorithm

The lifting algorithm [14] allows for an in-place computation of the wavelet transform. It makes use of three steps as shown in figure 4.3.

The first step is to split the input vector into even and odd indices. A vector "odd$^{(0)}$" takes all the odd numbered samples while "even$^{(0)}$" will take all the even ones. This process is also referred to as the lazy-wavelet.

Figure 4.2: *Two Dimensional DWT*

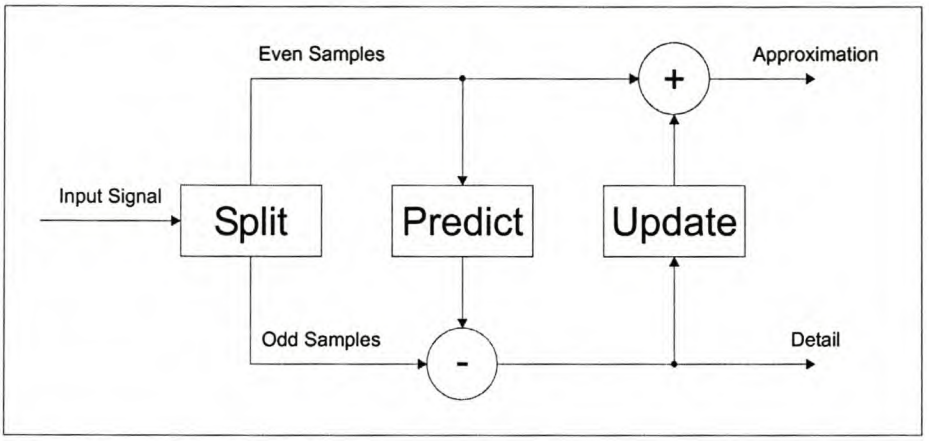

Figure 4.3: *The Lifting Scheme*

The predict phase uses a function that approximates the data set. The difference between the approximation and the actual data replaces the odd elements of the data set. The even elements are left unchanged and become the input for the next step in the transform. The predict step, where the odd value is "predicted" from the even value is described by the equation: $\text{odd}_i^{(1)} = \text{odd}_i^{(0)} - P(\text{even}_i^{(0)})$.

The predict phase is followed by the update phase. In the update phase the even elements are replaced with averages. The original value of the odd elements had been overwritten by the difference between the odd element and its even "predictor". So in calculating an average the update phase must operate on the differences that are stored in the odd elements: $\text{even}_i^{(1)} = \text{even}_i^{(0)} + U(\text{odd}_i^{(1)})$.

To reconstruct the original vector, the operation is reversed. First an undo-update, then undo-predict and finally a merge operation is computed. The algorithms are exactly the reverse of those used in the forward transform.

One of the big advantages of the lifting algorithm is its ability to map integers to integers. This is done by rounding the results of the prediction and update steps to the nearest integer. Rounding is a non-linear operation and thus it adds non-linearity to the transform (The non-linearity refers to the fact that the effect of the rounding is not a linear function of the input data), but it had been shown to still be perfectly reversible. The benefit from mapping to integers are two fold: First integers usually require less memory to store than floating point numbers and secondly, integer arithmetic usually executes faster than floating point arithmetic and requires less silicon space (resulting in power savings as well).

4.1.3 Wavelet Filters and Selecting Suitable Filters

So far only the Haar-wavelet has been mentioned but many different wavelet functions (or filters) exists. This section will show a few of the more common wavelets and list their properties.

4.1.3.1 The Haar-wavelet

See Figure 4.4. The Haar-wavelet is the simplest of all the wavelets. All it does is to compute average and difference coefficients from neighbouring data elements.

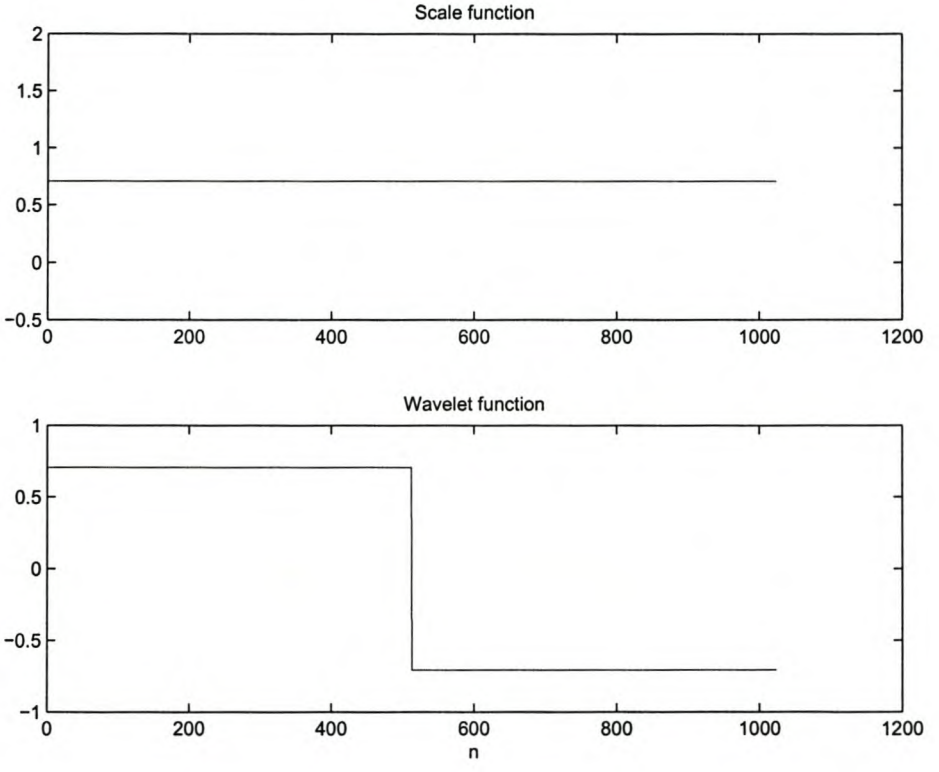

Figure 4.4: *The Haar-Wavelet*

The scaling function is produced by iteratively applying the low-pass wavelet filter on a discrete delta function an infinite number of times. The wavelet function (or mother wavelet) is produced similarly by iteratively filtering a discrete delta function with the high-pass wavelet filter.

4.1.3.2 The Daubechies 4-Tap Wavelet

See Figure 4.5. The Daubechies family is a family of asymmetrical orthogonal wavelets. They have the special property of having the maximum possible number of vanishing

wavelet moments. A smooth function can be approximated around a point using polynomials. A wavelet transform with vanishing moments is zero over polynomials. Thus, vanishing moments imply that the result of the transform of a smooth signal will be almost zero.

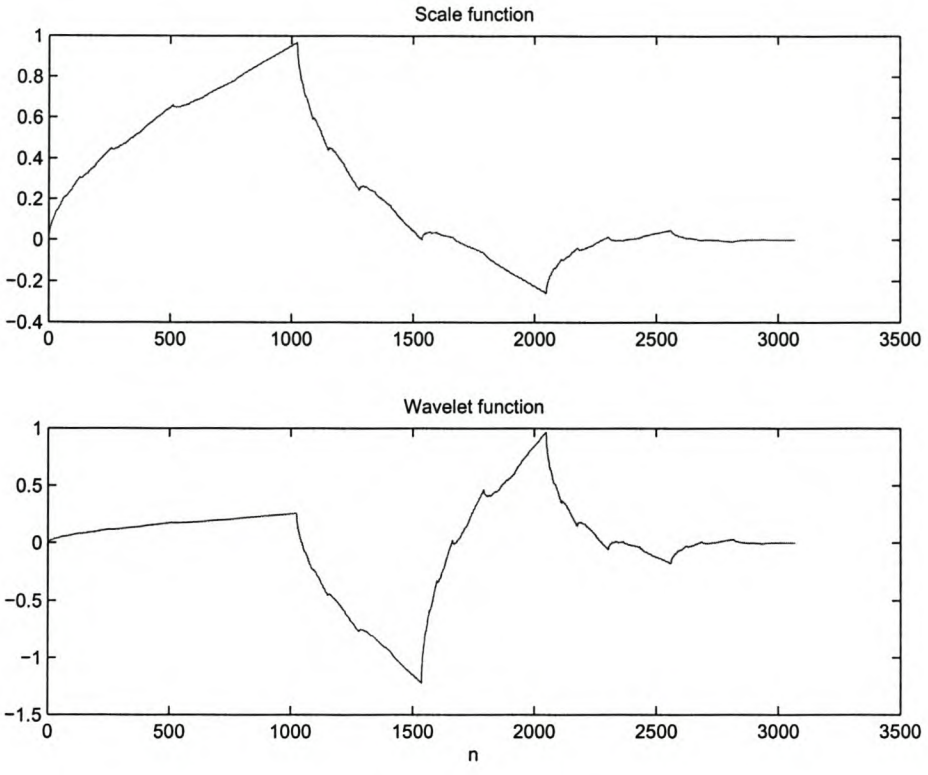

Figure 4.5: *The Daubechies 4-Tap Wavelet*

4.1.3.3 The (9,7) Biorthogonal Wavelet

See Figure 4.6. The biorthogonal family of wavelets are all constructed with linear phase (symmetric) filters. Linear phase filters yield the following advantages:

1. Linear phase filters allows the use of symmetric extension of the signal being transformed. This will reduce the edge effects.

2. Linear phase filters preserves the position of signal details which is of especial interest in image compression.

The (9,7) Biorthogonal wavelet is what is used by the American Federal Bureau of Investigation for their fingerprint compression system.

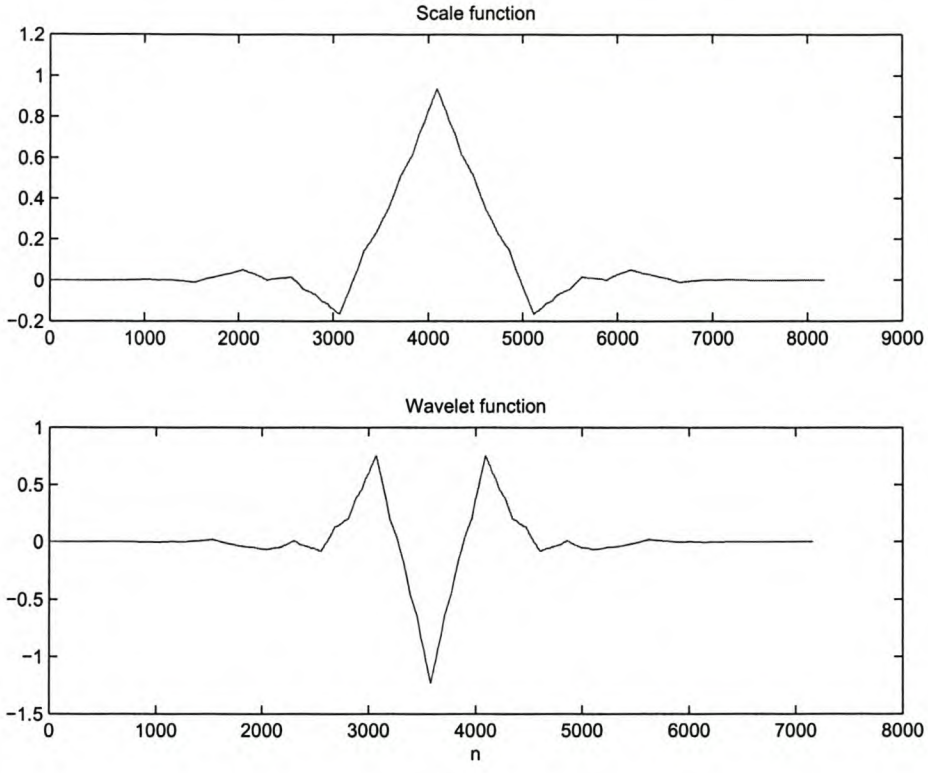

Figure 4.6: *The (9,7) Biorthogonal Wavelet (Analysis)*

4.1.3.4 Selecting Suitable Wavelet Filters

In the image compression system we will define a suitable wavelet as one that results in the highest image quality after the lossy image compression. Satellite images taken from different altitudes or with different spatial and spectral resolutions will have different characteristics. By selecting wavelets that have the same characteristics in their scaling and wavelet functions, the image compression will result in better image quality at the same compression ratio.

For image compression in general, the biorthogonal wavelet family (of which the (9,7) wavelet from the previous section is a member) is a good family. Biorthogonal wavelet filters are symmetric, and this property preserves the position of signal details, which is an important property for image compression. The (9,7) wavelet was found to be one of the wavelets that consistently delivers higher quality for a wide selection of images.

However, it is important to design the system to allow the wavelet filter to be changed after the mission launch. The best images to use in selecting a wavelet filter will be the images that will be compressed using that wavelet filter.

4.2 Embedded Zerotree Wavelet Encoding

This section looks at the EZW algorithm, the quantiser in the image compression system. The wavelet transform does not compress the data at all. Compression is the task of the quantiser (and thus the EZW algorithm) and also the entropy encoder.

4.2.1 Introduction

The Embedded Zerotree Wavelet (EZW) algorithm was designed by Jerome M. Shapiro [13] in 1993, specifically for implementation with the DWT. Its development was based on two observations:

- The larger a wavelet coefficient is, the more information it contains, and therefore the more important it is. The EZW algorithm therefore encodes the larger wavelet coefficients first.
- Maximum and average absolute coefficient values tend to get smaller as one moves from the lower frequency subbands (the highest scale) to the higher frequency subbands. This is illustrated in Figure 4.7. This observation leads the way to the formation of zerotree structures as described in Section 4.2.2.

The output produced by the EZW algorithm is progressive in nature: as more data is added to the compression process, the more detailed the reconstructed image will be. Progressive coding is also known as embedded coding - hence the "E" in EZW. The embedded property is of specific interest to the satellite environment. To decide whether it is worth downloading an image from the satellite, it is only necessary to download a part of it and decode that. If the image is of interest, the rest can be downloaded to add more detail to the image. This in itself may also save communication time between the satellite and the ground-station.

4.2.2 The Zerotree Structure

A zerotree structure is a tree in which the parent-object has four child objects. Each of these child objects in turn acts as a parent to four child objects, and so on. The absolute value of the objects in a zerotree structure decreases from parent to child. This is an important property for the EZW algorithm because, if a coefficient is found to be insignificant (see Section 4.2.3), all the children will be insignificant too, and the "branch" will be deemed to not contain any important information. In this way, a whole tree could be encoded as a single symbol, resulting in data reduction.

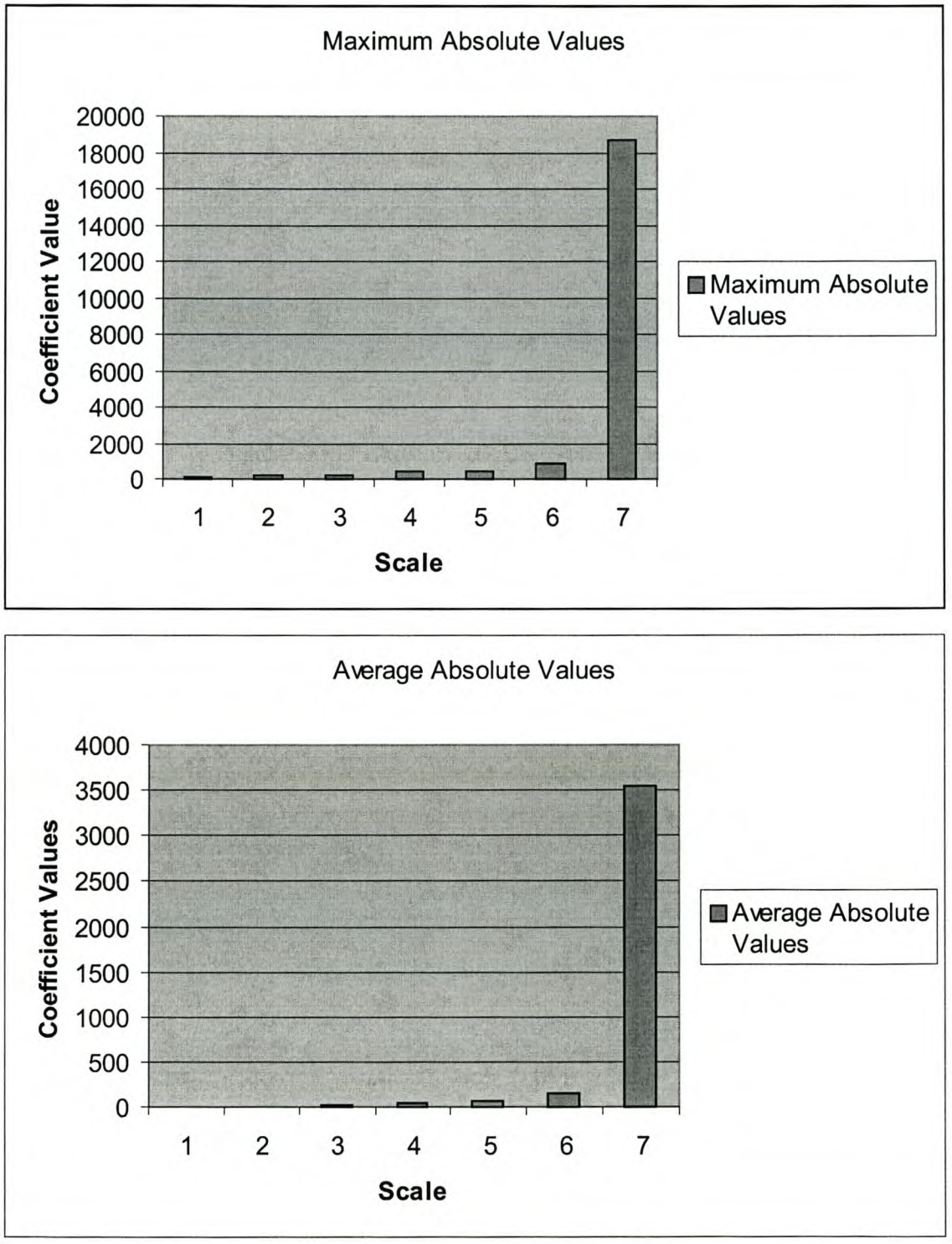

Figure 4.7: *Wavelet Coefficient Values across Wavelet Scales*

The coefficients in the lower frequency subbands can be thought of as having four children (or descendants) in the next higher subband. Each of these children will in turn have four children in the next higher subband. This is illustrated in Figure 4.8. Note the position of the children with respect to the parent. This is important for storing positional information of the coefficients in EZW encoding. EZW encoding, and the EZW algorithm in general, is described in the next section.

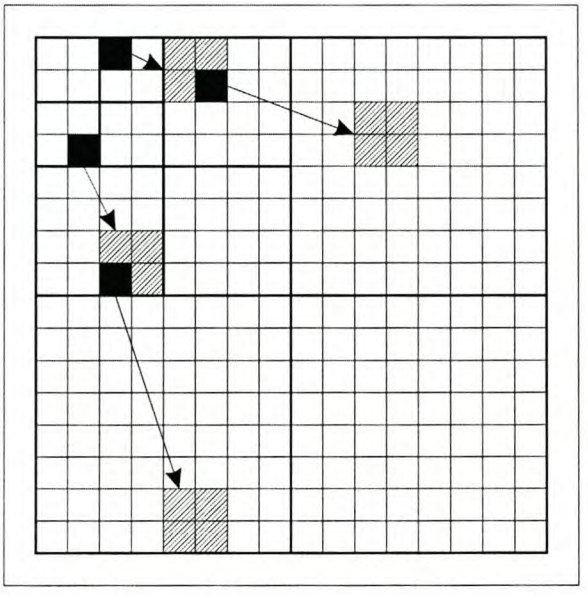

Figure 4.8: *Zerotree Structures in Wavelet-Transformed Images*

4.2.3 The EZW Algorithm

Given an image, the image compression program creates a two-dimensional DWT of the input image. This is passed to the EZW algorithm, performing the quantiser function, which calculates an initial threshold, using the following formula:

$$T_{initial} = 2^{\lfloor \log_2(k) \rfloor} \tag{4.2}$$

where $T_{initial}$ is the initial threshold, and k is the maximum absolute coefficient value. That is, the initial threshold is the largest power of two that is less than the maximum absolute value of the coefficients. Using this initial threshold, a 'dominant-pass' procedure (see Section 4.2.3.1 for detail) carried out on all the coefficients will identify all those coefficients that are significant; that is, all those coefficients that are larger in absolute value than the initial threshold.

After the dominant-pass procedure is completed, the current threshold is halved, and a subordinate-pass procedure (see Section 4.2.3.2) is performed. It progressively encodes

detailed information about the significant coefficients. This process (starting with a dominant-pass at the current threshold) is reiterated until some desired state is achieved (see Figure 4.9 for diagrammatic representation).

The 'desired state' is either when the threshold reaches a minimum value (usually one but it is possible to continue with threshold values between zero and one) or when a predetermined function is satisfied. The most common function is one counting the number of bits written to the compressed data stream and ending when it reaches a specified value. Since the input bit count is known and the number of output bits controlled, this function can guarantee exact compression ratios. Another possible function is an error function, which keeps track of the maximum absolute pixel error between the original and reconstructed images, and ends when this error falls below a specified margin.

4.2.3.1 The Dominant Pass

The Dominant Pass scans the coefficients at the current threshold level to identify significant coefficients and zerotrees. It outputs the following four symbols:

- **P** (Positive Significant) when a positive coefficient has become significant at the current threshold.
- **N** (Negative Significant) when a negative coefficient has become significant at the current threshold.
- **ZT** (Zerotree Root) when a coefficient and all its descendants (but not its parent) is insignificant.
- **IZ** (Isolated Zero) when an insignificant coefficient has at least one significant descendent.

These symbols are passed to the entropy encoder for the purpose of reducing the coding redundancies.

Figure 4.10 demonstrates how the dominant pass selects which symbol to output. The first step is to check whether the current coefficient was previously found to be significant in which case the coefficient is skipped and nothing is outputted. If the coefficient has become significant at the current threshold, the absolute value of the coefficient less the current threshold value, is appended to the subordinate list and either the positive significant (P) or negative significant (N) symbol is outputted. The subordinate list records all the significant coefficients. On the other hand, if the coefficient is not significant, the next step is to check if the insignificant coefficient is a child of an already discovered zerotree, in which case no symbol is outputted. If the coefficient is not part of an already

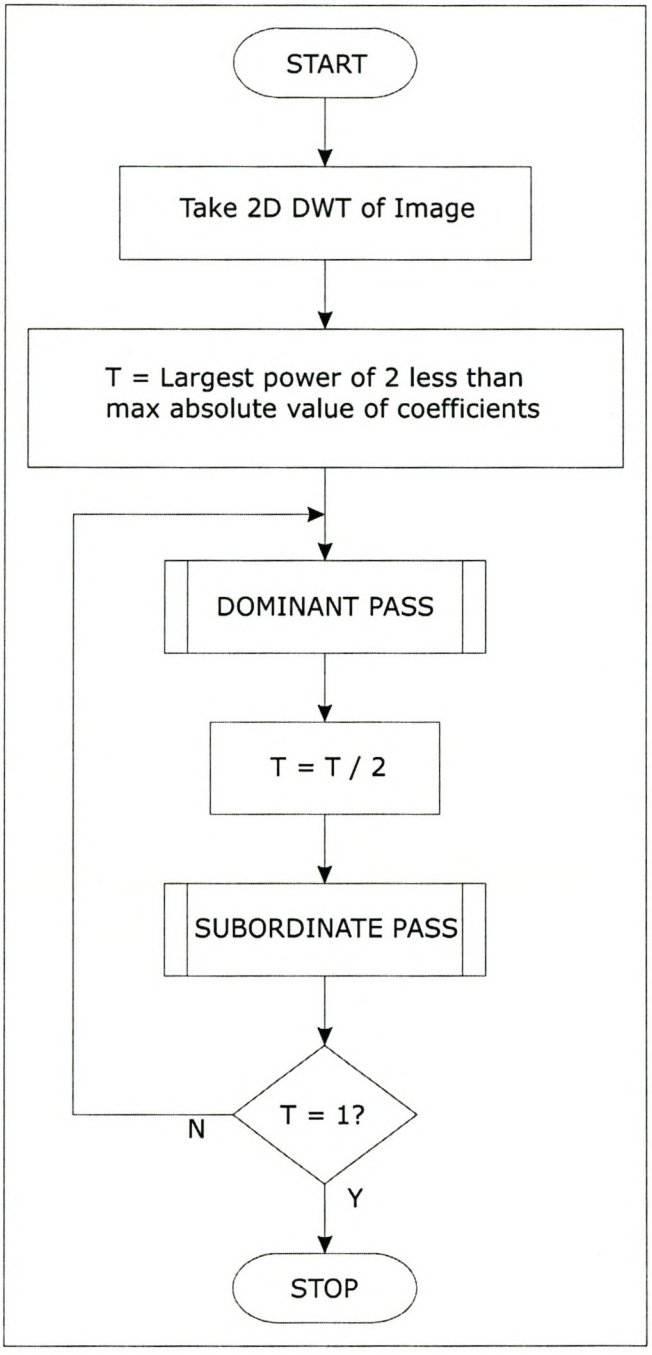

Figure 4.9: *Block Diagram of an Image Compression Program utilising the EZW Algorithm*

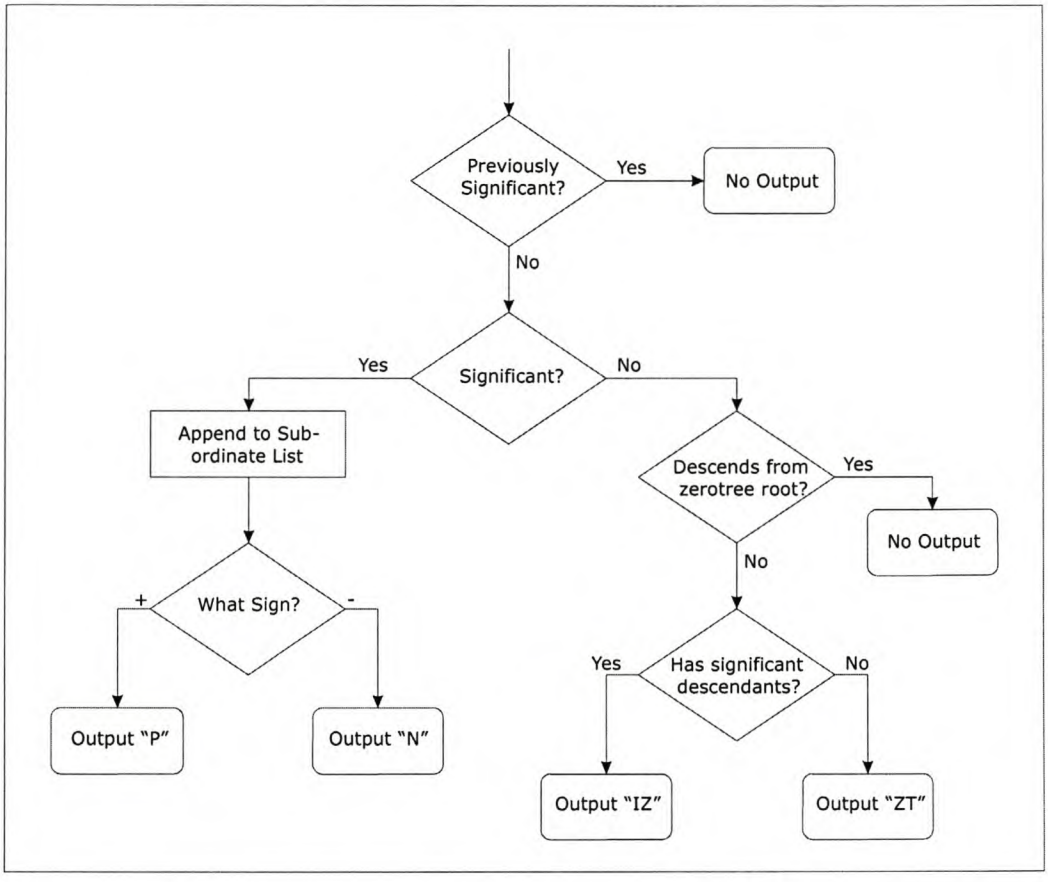

Figure 4.10: *Flowchart of the Dominant-Pass Procedure*

discovered zerotree, it must either be a root of a new zerotree or an isolated zero and the appropriate symbol is outputted.

The order in which coefficients are scanned is very important. Those in the lower subbands must be scanned prior to those in the higher subbands. The purpose of this ordering is to fully maximise the coding gain acquired from using the zerotree structure for the coding of insignificant coefficients. However, the scan order within subbands may vary. The most common scan order is the Raster-Scan order (Figure 4.11), where coefficients inside a subband are scanned in raster order (left to right and then top to bottom), and the lower subbands are always scanned first. Other scan orders can be used, on condition that the lowest frequency subbands are always scanned before the higher subbands.

4.2.3.2 The Subordinate Pass

The Subordinate Pass (sometimes called the Refinement Pass) 'refines' the value of each significant coefficient. For each coefficient in the subordinate list (coefficients are added by the dominate pass when they are found to be significant), the subordinate pass checks

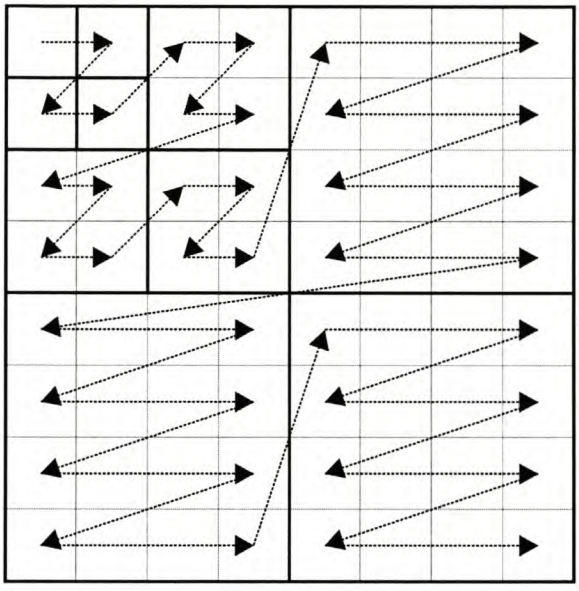

Figure 4.11: *Raster-Scan Order*

if their current value is larger or smaller than the current threshold value. If it is larger, a '1' is sent to the entropy encoder and the current threshold is subtracted from the coefficient value in the subordinate list. If the coefficient is smaller than the threshold, a '0' is sent to the entropy encoder. A block diagram of this procedure is shown in Figure 4.12.

Upon completion of the subordinate pass, the subordinate list should be sorted in order of highest to lowest values, in such a way that the decoder can reproduce the result. This is so that:

- The larger coefficients, which carry the most information, are in the front, and are thus coded first during successive subordinate passes;
- The entropy encoder will become more efficient if the subordinate pass generates a group of '1'-symbols followed by a group of '0'-symbols as apposed to randomly generating the symbols.

Since quick-sort is the fastest sorting algorithm available, it's inclusion here is recommended.

The subordinate pass may be left out of the algorithm. The purpose of the subordinate pass is to refine the already encoded coefficients and is not absolutely necessary. Although this will reduce the code size, execution time and memory requirements, it will adversely affect the quality of the reconstructed image. In one test the peak signal to noise ratio fell from 34.96dB to 17.60dB while keeping all other parameters the same (including the

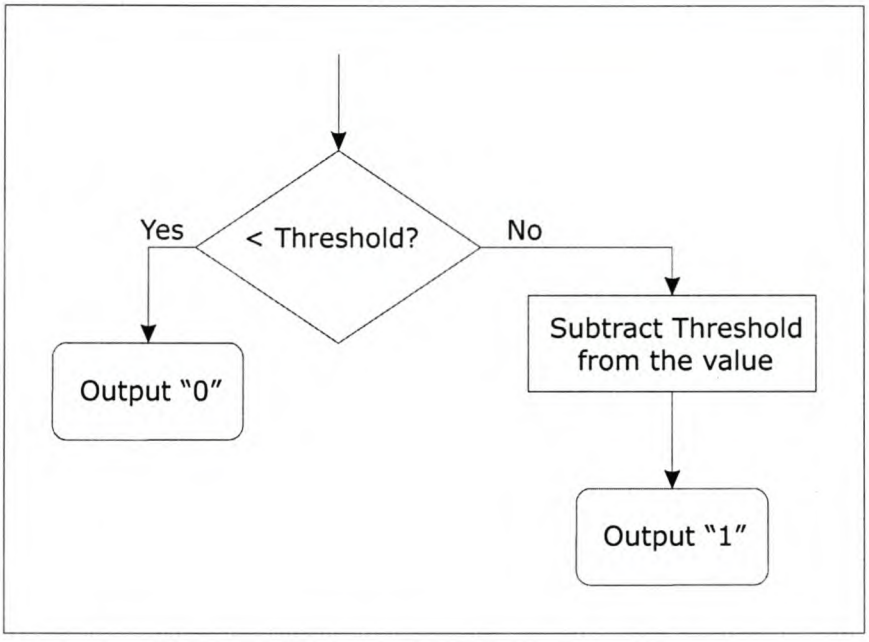

Figure 4.12: *Flowchart of the Subordinate-Pass Procedure*

target compression ratio). Due to this influence on the quality, it is not advisable to skip the subordinate pass. It is not possible to remove the dominant pass from the algorithm as this is the heart of the EZW algorithm.

4.2.4 Decoding the EZW Data Stream

Decoding the EZW data stream also uses dominant and subordinate passes. The dominant pass loads dominant-pass symbols from the data stream in order to retrieve significance information of the coefficients (all initially set to zero). It is important to use exactly the same scanning order as used during encoding to preserve the position of the reconstructed coefficients. When positive significant (P) and negative significant (N) symbols are encountered, the current coefficient's position is added to a new subordinate list. The dominant pass ends when all the coefficients in the image have been scanned.

The subordinate pass loads one bit from the data stream for each coefficient position stored in the subordinate list. This bit multiplied by the current threshold is added to positive and subtracted from negative coefficients.

Decoding will either stop when a predetermined cost function has been satisfied or when the data stream is empty.

There is a way to improve the image quality by changing the decoder. Given an original coefficient of 101, this will have been insignificant during a dominant pass with a threshold of 128 but is marked positive significant at a threshold of 64. The first subordinate pass

(with a threshold of 32) will see that $101 - 64 > 32$ and thus will encode a 1. The second subordinate pass (with a new threshold of 16) will encode a 0 because $101 - 64 - 32 < 16$.

The decoder will receive the positive significant symbol during the dominant pass with a threshold of 64. The decoder now knows that the original coefficient is larger or equal to 64 but definitely smaller than 128 because it is only now significant. During the first subordinate pass the decoder will receive a 1. This will tell the decoder that the coefficient's original must have been at least $64 + 32 = 96$ but still smaller than 128. The second subordinate pass will receive a 0. The zero tells the decoder that the original coefficient minus the current value of 96 was smaller than 16 (the current subordinate pass's threshold). That will move the upper limit of 128 down to 112. If the decoding ends here, the decoder knows the original coefficient's value was in the interval $[96, 112)$. By selecting the average of these two bounds (104 in this case), the average coefficient error (or quantisation noise) will be smaller which will result in a higher image quality after the reverse wavelet transform.

Figure 4.13 contains a plot of image quality versus compression ratio when using the lower bound, upper bound and average of the two bounds for calculating coefficient values. It is clear that using the average of the two bounds (or centre point) results in the higher image quality.

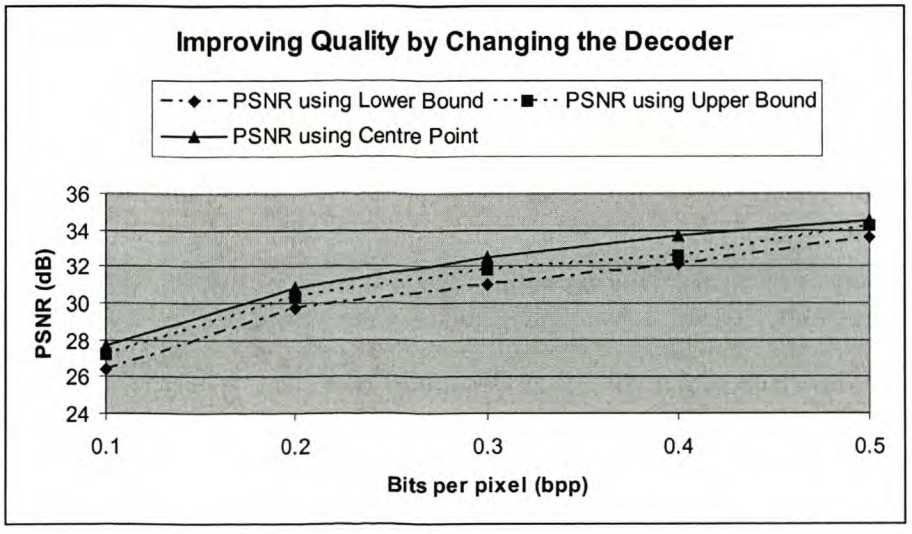

Figure 4.13: *Improving Image Quality by Tweaking the Decoder*

4.3 The Entropy Encoder

The task of an entropy encoder is to reduce coding redundancies. Coding redundancies are best explained by an example: In the English language the letter 'q' is almost always

followed by the letter 'u'. These two letters can therefore be stored as a single symbol.

As was stated in Chapter 3, the entropy encoder used in this project is the RangeCoder. The RangeCoder is an implementation by Michael Schindler [12] based on a paper presented in 1979 by G. Martin [8]. The RangeCoder is similar to the better known Arithmetic Coder and delivers almost the same compression as the Arithmetic Coder. However, it is faster than the Arithmetic Coder [12].

The RangeCoder was implemented with an adaptive quasi-static model also by Michael Schindler. This model allows the entropy encoder to adapt when the occurrence probabilities of the symbols generated by the EZW algorithm change.

For the implementation options discussed in the rest of this document, the RangeCoder and quasi-static model is considered an integral part of the EZW algorithm and will not be treated separately.

4.4 The EZW Algorithm's Influence on Images

The EZW Algorithm tends to act like a low-pass filter. This can be seen from the operation of the EZW algorithm. The algorithm encodes the largest coefficients first. The coefficients in the high frequency subbands tend to have the lowest absolute values (See Figure 4.7) and are therefore encoded last. When the EZW algorithm has filled its byte budget, it is these small coefficients in the high frequency subbands that are not encoded.

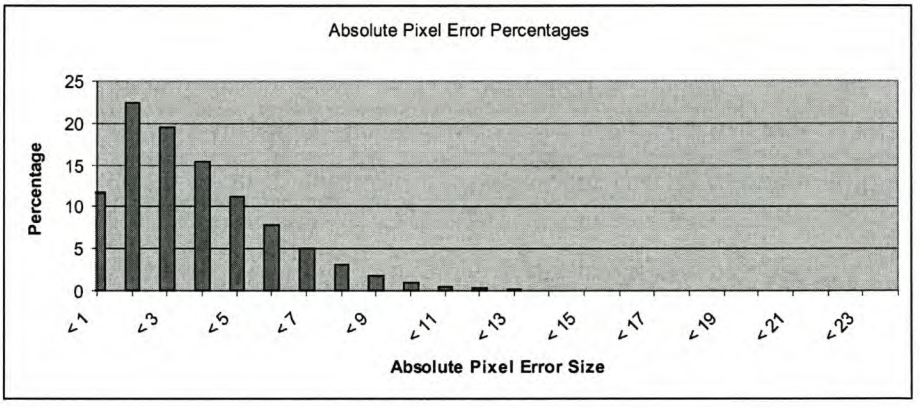

Figure 4.14: *A Typical Histogram of Absolute Pixel Errors Caused By EZW Compression*

Since these high frequency coefficients are small, the pixel errors introduced by not encoding them are also small. Figure 4.14 shows a histogram of the spread of pixel errors. Due to these small pixel errors, the EZW algorithm does not have a significant impact on

ratio-images. However, even small pixel errors can have large influences on the results of digital image classification procedures.

4.5 Algorithmic Complexity

The lifting implementation of the (9, 7) Biorthogonal wavelet (see Section 6.1) requires four multiplications and eight additions per pixel per wavelet scale. Equation 4.3 gives the number of multiplications required to calculate m wavelet levels for a $N \times N$-pixel image.

$$\text{Multiplications} = 4\sum_{i=0}^{m}\left(\frac{N}{2^i}\right)^2 \tag{4.3}$$

Twice that number of additions are required. The number of operations is proportionate to the number of pixels. For the discrete cosine transform the complexity is a function of $N \log(N)$. The DWT is therefore less complex than the DCT.

The EZW algorithm's (and also the RangeCoder's) number of calculations and thus also execution time, is highly dependent on its input. It is therefore impossible to calculate a complexity level for the whole algorithm. One idea for evaluating the algorithm on specific target platforms is given in Chapter 7 where detailed implementations of the EZW algorithm are discussed.

In the next chapter the implementation options for the whole system will be considered.

Chapter 5

Implementation Options

Having presented the theory of the wavelet transform and EZW algorithms, it is now possible to consider various implementation options for the image compression system. In this chapter, each option is presented, and it's advantages and disadvantages discussed.

When considering selection of an option for implementation, the environment (that of a satellite) in which it will function needs to be taken into account. Considerations include extreme temperatures (although temperatures are usually controlled effectively by the thermal subsystem), and radiation damage. Furthermore, micro-satellites are dependent on their solar arrays for all their power requirements. Power usage should preferably be kept below 10 watts.

5.1 Image Size and Memory Requirements

To calculate the two dimensional wavelet transform, the full image must be stored locally. This is because the wavelet transform operates on both the rows and the columns. Therefore, the system that will run the wavelet transform must have enough RAM to store the full image.

Using smaller images would reduce the size of memory required. Also, if larger images were split into smaller images, they could be compressed in parallel, thus improving the data throughput of the system. However, breaking the images up into smaller images reduces the effectiveness of the EZW compression, and may also cause blocking artefacts when the images are recombined.

Table 5.1 below lists the RAM requirement of the EZW algorithm for different image sizes.

Experiments showed that the EZW algorithm performs well with images of 512×512 pixels and larger. Only a small amount of RAM is required to process images of 512×512 pixels.

Image Size	Memory Requirement
128×128	$<$ 128kB
256×256	$<$ 512kB
512×512	$<$ 2MB
1024×1024	$<$ 8MB
2048×2048	$<$ 30MB
4096×4096	$<$ 120MB
8192×8192	$<$ 460MB

Table 5.1: *Memory Requirements for different Image Sizes (16bit Wavelet Coefficients)*

5.2 Integer vs Floating Point Wavelet Transform

One consideration when implementing the wavelet transform was whether to use integer or floating point arithmetic.

Floating point arithmetic may be implemented either at the hardware level or at the software level. A software implementation of floating point arithmetic will have significant slower performance than a hardware implementation. The reason for this is that many instructions must be executed in order to perform a single floating point operation. Moreover, systems with hardware floating point support will typically run the floating point unit and the integer unit in parallel, thus resulting in a higher system peak performance. From this can be concluded that hardware floating point support should be favoured above a software implementation.

Despite using floating point arithmetic, the wavelet transform (using the lifting algorithm) may also be implemented using integer arithmetic. The $(9, 7)$-Biorthogonal wavelet transform can be implemented using 16 bit integer arithmetic. Integer arithmetic has the following advantages over floating point arithmetic:

1. Integer arithmetic is simpler. Floating point values contain three separate parts: the fraction, the exponent and the sign. Integers stored in two's complement form are single entities which greatly simplifies the arithmetic. The result is that an integer unit occupies less silicon space, and operates at higher speeds.

2. Most hardware floating point units follow the IEEE 754 standard. This standard supports four floating point formats, the smallest of which is 32 bits wide. Storing floating point numbers would therefore require more memory than storing the 16 bit integers would.

The conclusion of the research is that an integer implementation will result in a higher

performance system, requiring less memory and power than an equivalent floating point implementation system would.

5.3 Hardware Implementation Options

In this section, a number of processor options are presented for consideration. They include implementation using: a single processor for the whole system; a general purpose processor and a DSP chip; a general purpose processor and dedicated wavelet hardware in FPGA; dedicated wavelet hardware and a soft-core processor; and finally, an HDL implementation of both the wavelet and EZW algorithms.

5.3.1 Single Processor

Research was performed to determine the possibility of using a single processor to support the wavelet transform, the EZW algorithm and the entropy encoder. In order to perform these tasks in the limited time, a fast processor is required. The desktop PC used to develop the wavelet transform and EZW software required approximately half a second to run the program. This time includes operating system overheads, as well as extra timing routines added by the GNU C compiler and is not an accurate indication of the execution speed.

The only sufficiently fast processors are the higher end desk-top PC (and similar) processors. The downside to using such processors is their power consumption. For example, the power consumption of high end Intel Pentium 4 processors is between 50 and 80 watts, while the high end AMD Athlon XP processors consume between 60 and 75 watts. All these processors have been designed for terrestrial use where power consumption is generally not an issue.

Not only is this level of power consumption very high for micro-satellites, but the heat generated as a result will be problematic on satellites. The single processor option is therefore not viable for micro-satellite usage.

5.3.2 General-Purpose Processor and DSP Chip

The second option considered was to use a fast processor for running the EZW and entropy encoding algorithms, and a separate digital signal processing (DSP) chip, with wavelet support, to perform the wavelet transforms.

Digital signal processors are ideally suited for applications where a small window of input data is manipulated by a small algorithm. This is exactly what the wavelet

transform is, as it operates on only a few neighbouring pixels at any one time, and runs the same small algorithm on all input data.

5.3.3 General-Purpose Processor and Dedicated Wavelet Hardware in a FPGA

Instead of using a DSP chip to support the wavelet transform, as suggested in the previous implementation option, the algorithm can be written in VHDL (Very high speed integrated circuit Hardware Description Language), and programmed in a FPGA. One advantage gained in FPGAs is the ability to do things in parallel which is not possible in processors.

In the past it was difficult to implement multiplication routines in FPGAs because of the amount of logic cells required to do so. However, current FPGAs have considerably more logic cells than before (Altera's Stratix family have between 10000 and 114000 logic elements), and a number of them have hardware multiplier blocks included (like Altera's Stratix and also Xilinx's Virtex II) as well. The newer FPGAs are also capable of running at significantly higher clock frequencies than before. For example, Altera reports a maximum frequency of 420MHz for the Stratix family.

A suitable evaluation board with a Stratix FPGA, with sufficient onboard memory, was available for use in the project.

5.3.4 Dedicated Wavelet Hardware and Soft-Core Processor

A soft-core processor is a VHDL (or verilog or any other hardware description language) model of a microprocessor, which, can run in a configurable logic device such as an FPGA.

The wavelet transform and EZW algorithm were run on Altera's NIOS soft-core processor running at 50MHz in a Stratix FPGA. Both algorithms required between 50 and 60 seconds to complete.

The conclusion of this section of the research is that soft-core processors are too slow to meet the data throughput requirement.

5.3.5 Hardware Description Language Implementation of the Whole System in a FPGA

The final option researched is that of implementing the wavelet transform, the EZW algorithm and the entropy encoder with an HDL. The FlexWave-II system utilises an HDL implementation of an algorithm similar to EZW.

Converting from C-source code to an HDL implementation has become easier since the introduction of automated tools for this purpose.

5.4 Conclusion

The use of a single processor for the whole system was found to be impractical. This is due to the fact that the power requirements of high performance processors was found to be too high for micro-satellites.

Due to the availability of a Stratix-FPGA evaluation board, the VHDL option for implementing the wavelet transform was favoured above the DSP option. An integer implementation of the wavelet transform is discussed in Chapter 6.

Converting the EZW algorithm to VHDL is an interesting solution and is a possible future research project, but it was decided to first investigate the implementation of the EZW algorithm and entropy encoder on a general purpose processor. Such a processor can potentially also be used for other purposes on the satellite (such as a backup OBC).

An image size of 512×512 pixels was selected for the system. This is because images with square dimensions, possessing side lengths that are a power of two, permit simplifications of both the wavelet transform and the EZW algorithms, thus improving overall system performance. Another reason for selecting an image size of 512×512 pixels is that this size does not require excessive onboard memory.

Chapter 6

VHDL Wavelet Implementation

This chapter presents an implementation of the lifting algorithm for calculating the wavelet transform, in VHDL. A possible full wavelet transform system design will also be discussed.

6.1 From the Lifting Algorithm to a Wavelet Transform Pipeline

Any discrete wavelet transform can be decomposed into a finite sequence of simple filtering steps, called lifting steps [6]. Using this method, it can be shown that the popular $(9,7)$ Biorthogonal Wavelet factorises to the following implementation:

$$\begin{aligned} s_i^{(0)} &= x_{2i} \\ d_i^{(0)} &= x_{2i+1} \end{aligned} \tag{6.1}$$

$$d_i^{(1)} = d_i^{(0)} + \alpha \times \left(s_i^{(0)} + s_{(i+1)}^{(0)}\right) \tag{6.2}$$

$$s_i^{(1)} = s_i^{(0)} + \beta \times \left(d_i^{(1)} + d_{(i-1)}^{(1)}\right) \tag{6.3}$$

$$d_i^{(2)} = d_i^{(1)} + \gamma \times \left(s_i^{(1)} + s_{(i+1)}^{(1)}\right) \tag{6.4}$$

$$s_i^{(2)} = s_i^{(1)} + \delta \times \left(d_i^{(2)} + d_{(i-1)}^{(2)}\right) \tag{6.5}$$

$$\begin{aligned} s_i &= \zeta s_i^{(2)} \\ d_i &= \frac{d_i^{(2)}}{\zeta} \end{aligned} \tag{6.6}$$

Symbol	Value	Fixed Point Representation	Shifted
α	−1.586134342	−1624	10
β	−0.05298011854	−217	12
γ	0.8829110762	904	10
δ	0.4435068522	908	11
ζ	1.149604398	N/A	N/A

Table 6.1: *Constants for the (9,7) Biorthogonal Wavelet*

The values of the constants (α, β, γ, δ and ζ) are presented in the "Value" column of Table 6.1.

Equations 6.1 split the even and odd samples. It is known as the split, or lazy wavelet, phase. The filtering is performed by Equations 6.2 through 6.5, while Equations 6.6 take care of the normalisation. Normalisation ensures that the average value of the low frequency signals remain the same as in the original signals. This step is not essential and, since it involves additional multiplications, it was decided not to implement it.

The constants can be implemented as fixed point numbers (as shown in Table 6.1), resulting in an integer to integer transform.

Each function operates on the results of the previous two functions. Thus they all run sequentially. An important observation is that they can run in pipeline: To calculate the i-th iteration of any of the functions, the result of two consecutive iterations of the previous function is required. Therefore each of the four filter functions need only wait for two elements of the previous filter function to complete before they can begin their execution. This assumes that the original input data had already been split into the $s^{(0)}$ and $d^{(0)}$ vectors.

Using this observation, the transform can be implemented in a pipeline, as illustrated in the next section.

6.2 The Pipeline Design

The VHDL source files of the wavelet pipeline can be found in Appendix A.

Assume that the input image line has already been split into the even and odd samples (the s and d vectors). Then each of the four filtering functions can be implemented in a manner similar to the diagram for the first function (the so-called alpha stage because of the α-constant in the function), shown in Figure 6.1.

The "Wait Add" block's output is always the sum of the current and previous samples, and thereby calculates $s_i^{(0)} + s_{(i+1)}^{(0)}$ in the case of the first function. The next block

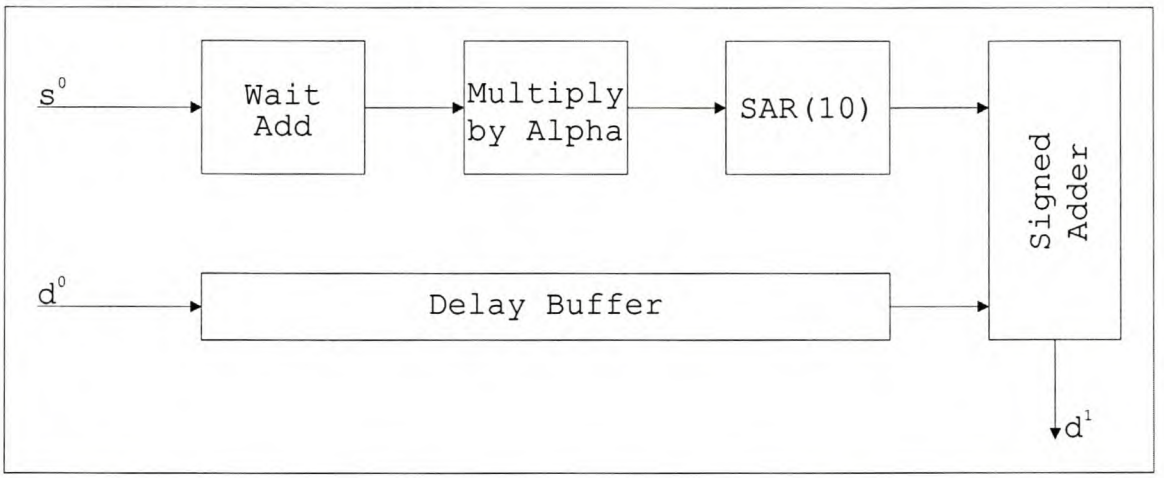

Figure 6.1: *Alpha Stage of the Wavelet Pipeline*

- "Multiply by Alpha" - multiplies that result with a constant (α in this case). The resultant is then arithmetically shifted right, as indicated by the "SAR(10)"-block in the diagram, by the number of bits as shown in the "shifted"-column of Table 6.1. The next step is to add that result to the other input stream ($d^{(0)}$ for the first function).

Due to delays (or latency) incurred by the adding, multiplying and shifting, a delay buffer (implemented as a FIFO-buffer) is required to ensure that the correct sample is added to the result. The final output of the alpha stage is the $d^{(1)}$ data stream.

This method is ideal for infinite length data streams, but in terms of the current application, finite data streams are required. When calculating either $d^{(1)}$ or $d^{(2)}$, the formula requires an $(i+1)$-th element. To take this boundary condition into account, Equations 6.2 and 6.4 are modified as shown in Equations 6.7 and 6.8.

$$d^{(1)}_{i=N-1} = d^{(0)}_{(N-1)} + \alpha \times \left(2 \times s^{(0)}_{(N-1)} - s^{(0)}_{(N-2)}\right) \tag{6.7}$$

$$d^{(2)}_{i=N-1} = d^{(1)}_{(N-1)} + \gamma \times \left(2 \times s^{(1)}_{(N-1)} - s^{(1)}_{(N-2)}\right) \tag{6.8}$$

A similar boundary condition is encountered when calculating the first element of either $s^{(1)}$ or $s^{(2)}$. Their formulas require an $(i-1)$-th element. Equations 6.9 and 6.10 show the solution to this boundary condition.

$$s^{(1)}_{i=0} = s^{(0)}_0 + \beta \times \left(2 \times d^{(1)}_0\right) \tag{6.9}$$

$$s^{(2)}_{i=0} = s^{(1)}_0 + \delta \times \left(2 \times d^{(2)}_0\right) \tag{6.10}$$

An alternative solution to using conditional formulas is to edit the data streams by adding extra data elements to the data stream. This is done as follows: when calculating

the d-stream, an extra element is added to the input s-stream, located at the end of the data stream. The value of this element is $s_{i=N} = s_{N-1} - s_{N-2}$. By using the original formula, with this extra data element to calculate the last element of the d-stream, one can obtain exactly the same result as that obtained by Equation 6.7.

A similar technique is used for calculating the s-streams. An extra element must be inserted at the front of the input d-streams with the same value as the first element of the stream. Adding these two elements together will give the $2 \times d_0$, as required by Equation 6.9.

Figure 6.2 shows a diagram of the full pipeline. The beta, gamma and delta stages are all basically the same as the alpha stage shown in Figure 6.1, but the multiplier and the shift constants differ. The "Edge Generator"-blocks are used to add the extra data elements required to fix the boundary conditions at the end of s-data streams. The "duplicator"-blocks duplicate the first element of the d-data streams to fix that boundary condition. These two components are discussed in more detail in sub-sections 6.2.1 and 6.2.2. Some extra delay buffers are required to ensure that all the data streams are synchronised in the wavelet pipeline. For simplicity, not all these delay buffers are shown in the diagram.

6.2.1 The Edge Generators

An edge generator calculates the difference between the previous two input samples, and places either this difference, or the current input sample, onto the output data stream. See Figure 6.3 for a diagram of the edge generator.

On each rising clock edge, the previous sample, which is stored in LatchA, is moved into LatchB, overwriting its contents. The current sample is then moved into LatchA. The subtractor then subtracts the contents of LatchB from that of LatchA, and makes the resultant available to the multiplexor (MUX).

A trigger input is used to determine whether the subtractor's output or the current input sample is outputted.

6.2.2 The Duplicators

Figure 6.4 shows the diagram of the duplicator. The duplicator latches the input, and a multiplexor selects between the stored (previous) sample and the current input sample, based on the control input signal.

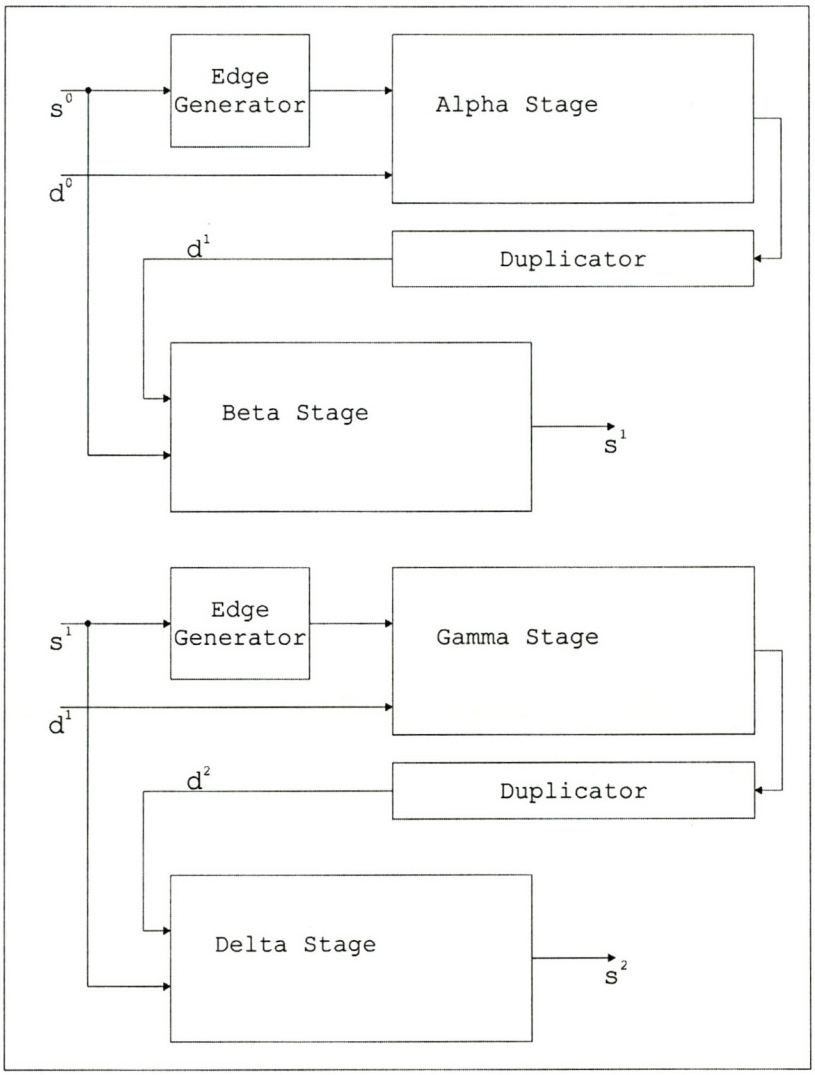

Figure 6.2: *The Wavelet Pipeline*

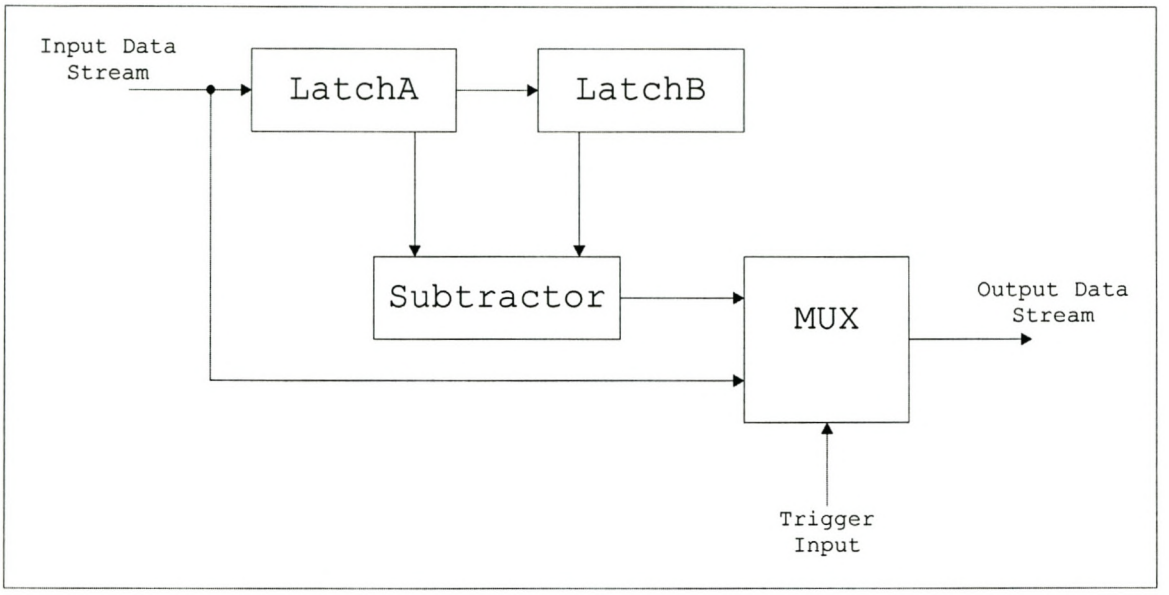

Figure 6.3: *The Edge Generator*

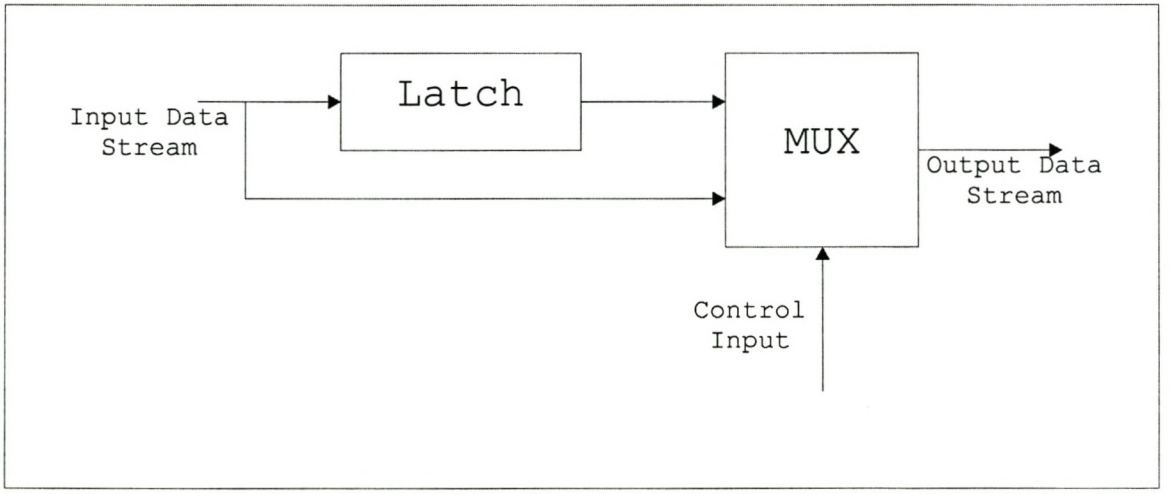

Figure 6.4: *The Duplicator*

6.3 Quartus Simulation of the Wavelet Pipeline

For the implementation of the wavelet pipeline, Altera's Quartus II version 3.0 software was used to compile and simulate the wavelet pipeline design. Table 6.2 lists the results of the compilation.

Device	EP1S25F672C6
Total logic elements	539 / 25,660 (2%)
Total pins	70 / 473 (14 %)
Total memory bits	528 / 1,944,576 (< 1 %)
DSP block 9-bit elements	8 / 80 (10 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)

Table 6.2: *Quartus Compiler Results*

The maximum internal clock frequency is reported as 241.72MHz between source register "MulDeltaIn[7]" and destination register "SAR3In[23]".

Figures 6.5 and 6.6 show the simulation results. The output data (signals "sf" and "df") are valid from 315ns until 630ns.

As can be seen on the simulation, the wavelet pipeline requires five input control signals ("aclr", "eGen0Ctrl", "eGen1Ctrl", "dub0Ctrl" and "dub1Ctrl") which must be triggered at the correct times. In addition, there is no indication as to when the output data will be valid. Therefore the system which is connected to the pipeline must know when to start reading the output. In order to reduce these complexities for the system using the wavelet transform, at a minimum, a control unit must be added to the wavelet pipeline.

The estimated power consumption of the simulation shown in Figures 6.5 and 6.6 is provided in Table 6.3.

The next section will describe one possible implementation of the wavelet pipeline in a full wavelet transform system.

6.4 Implementation of the Wavelet Pipeline, and Results

In this section a possible implementation of the wavelet pipeline of Section 6.2 is suggested. A timing analysis is also performed with the aim of estimating the performance level that is possible with this implementation.

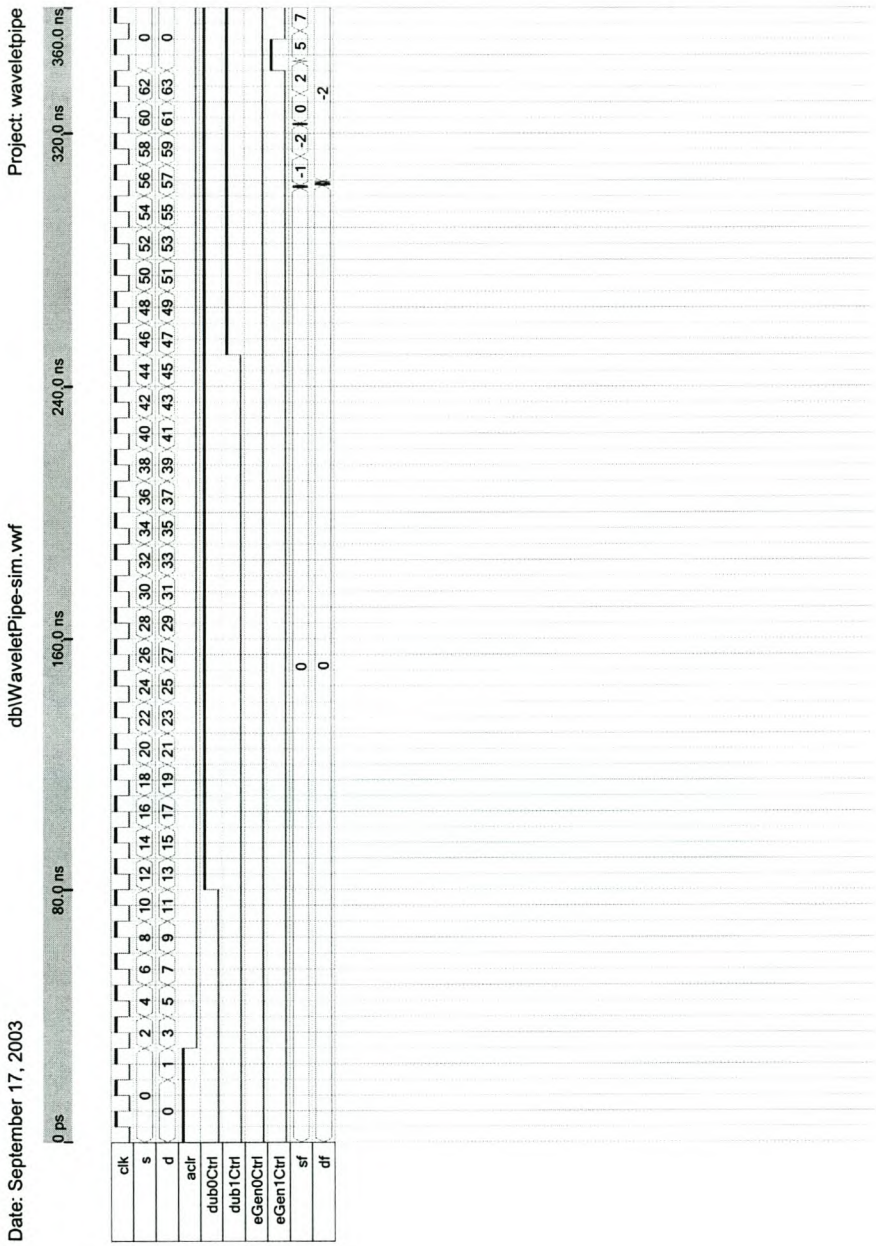

Figure 6.5: *Quartus Simulation Results (1/2)*

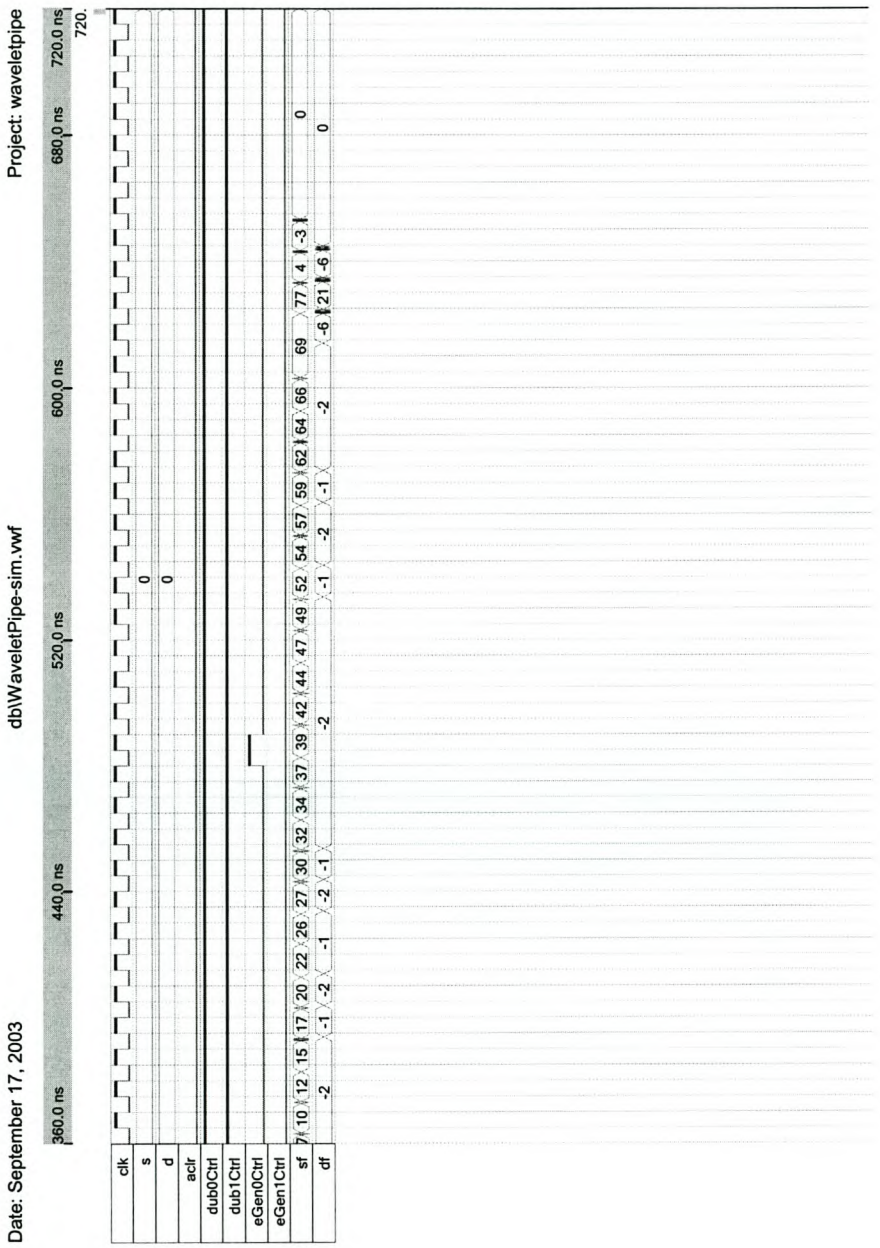

Figure 6.6: *Quartus Simulation Results (2/2)*

Power estimation start time	0 ps
Power estimation end time	720 ns
Total Internal Power	354.80 mW
Total Standby Internal Power	281.25 mW
Total Logic Element Internal Power	18.93 mW
Total IO Buffer Internal Power	0.56 mW
Total M512 RAM Internal Power	0.66 mW
Total M4K RAM Internal Power	2.83 mW
Total Clocktree Internal Power	44.85 mW
Total DSP Internal Power	5.72 mW
Total IO Power	62.20 mW
Total IO Buffer Power	62.20 mW
Total Power	417.00 mW

Table 6.3: *Quartus PowerGauge Results*

6.4.1 A Suggested Implementation

A suggested implementation is shown in Figure 6.7. Image data is received via an input communications subsystem, and, is fed through a multiplexor (MUX) to the SDRAM controller. All data transfers between this system and the external SDRAM are done in a striping mode (or burst read/write mode) to ensure maximum memory data throughput. The image data is then read from the SDRAM, and sent through a splitter unit, that splits the input data into the s and d streams which are passed to the wavelet pipeline.

The wavelet pipeline's output goes through two delay (FIFO) buffers. SDRAM is typically single-port memory. This means there is only one address/data port which performs all read and write operations. The delay buffers are required to temporarily store the data while the input data is still being processed. Due to the fact that the two output streams should be stored separately, the one delay buffer (most likely the one storing the output d stream) will need to be longer than the other. This will ensure that the other stream can be completely copied to the SDRAM before the second stream is sent to the SDRAM.

The output from the SDRAM controller is also connected to an output communication unit. This output communication unit will send the results of the wavelet transform to the system running the EZW algorithm.

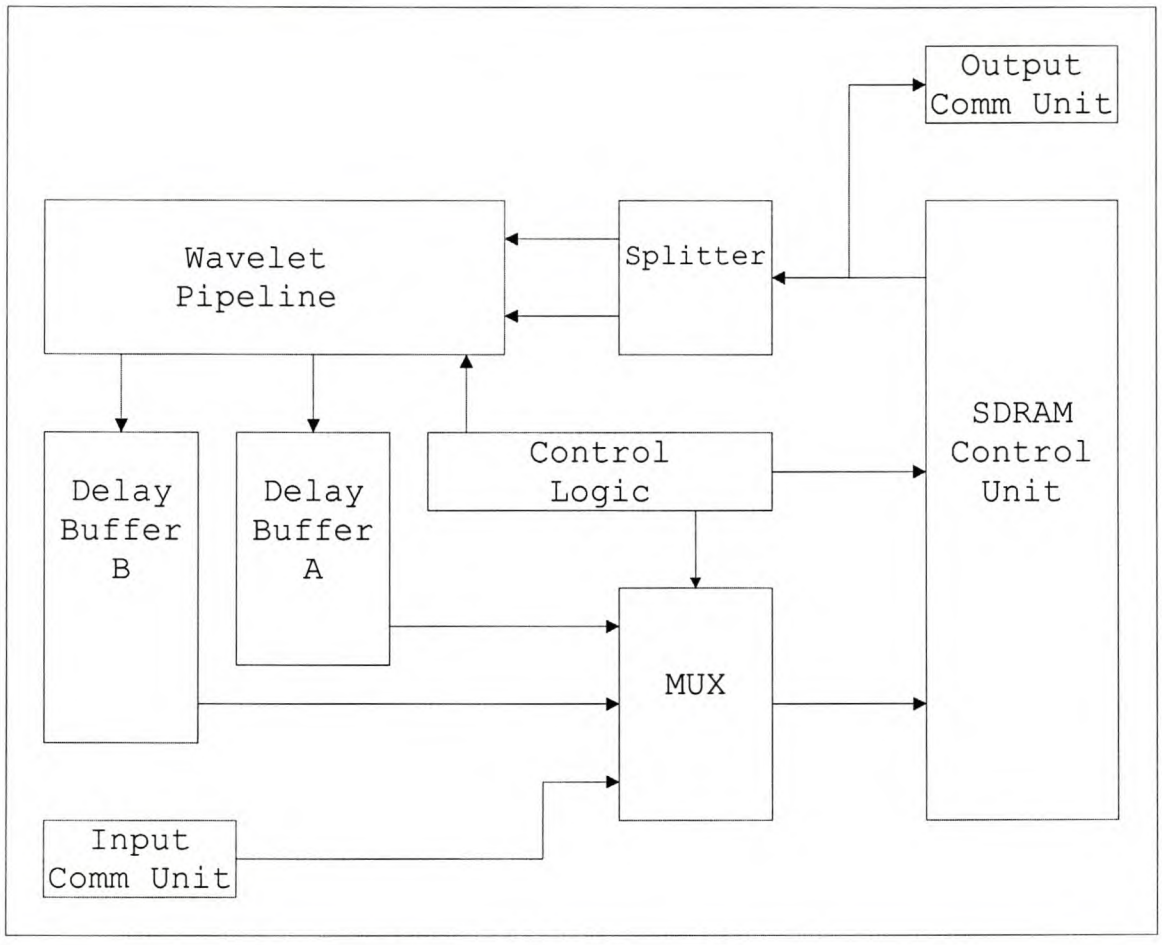

Figure 6.7: *An Implementation of the Wavelet Pipeline*

6.4.2 Operating Speed

SDRAM can typically run at 133MHz, which would ordinarily require that all subsystems in the wavelet transform implementation also run at 133MHz. However, since the data bus width is 32bits on the SDRAM side (SDRAM ICs typically have 32bit data busses), and only 16bits (The wavelet coefficients are stored as 16bit words) inside the wavelet transform unit, all the subsystems must effectively run at double the SDRAM clock speed, with the exception of the wavelet pipeline. That is because the splitter effectively halves the data rate again, since it splits the input data stream into two separate streams.

The compiler results of Section 6.3 indicate a maximum internal clock frequency of 241.72MHz. At this speed the wavelet pipeline can run fast enough for an SDRAM clock of 133MHz. However, the other subsystems must run at double the SDRAM clock. These subsystems will have to be carefully designed to run at a clock speed of 266MHz, but, according to the Stratix data sheet [4], such a high clock frequency should be possible.

The specifications for Micron Technology's 64Mbit SDRAM [9] show a configuration of

4 banks each with 2048 rows by 256 columns, having a data depth of 32bits. It is possible to read/write at maximum speed to any column or bank, as long as the row number stays unchanged. Since each wavelet coefficient is only 16bits wide, two coefficients can be stored in every memory position. This addressing configuration enables an image row to be stored in a single SDRAM-column. In order to optimise the image-column read/write speed, it is necessary to store four consecutive image columns in the four banks at one SDRAM-row address, and the next four image columns in the four banks at the next SDRAM-row address (see Table 6.4).

Image	SDRAM		
Row Number	Row Number	Column Number	Bank Number
0	0	0-255	0
1	0	0-255	1
2	0	0-255	2
3	0	0-255	3
4	1	0-255	0
5	1	0-255	1
...	...	...	...

Table 6.4: *Image Rows in the SDRAM Row/Column/Bank-structure*

With this configuration the read/write access latencies are as follows:

- Reading an Image-Row: This requires three cycles to activate the SDRAM-row, three cycles for the CAS-latency (Column Address Strobe Latency) - the time from when the column address is supplied until the data is ready - and three cycles for the precharge after the data has been read. The total overhead is nine cycles.

- Reading an Image-Column: Every four pixels (stored in four banks) in the column will require the full read overhead of nine cycles. The total overhead is $9 \times \frac{N}{4}$ where N is the number of pixels in the column.

- Writing an Image-Row: Again three cycles are required to activate the SDRAM-row and also three cycles for the precharge. The CAS-latency is not applicable to write operations. The total overhead is only six cycles.

- Writing an Image-Column: As with reading a column the overhead must be added for every group of four pixels. The total overhead is $6 \times \frac{N}{4}$ where N is the number of pixels in the column.

6.4.3 Data Throughput Analysis

All calculations assume the image size to be 512×512 pixels (eight bits each), and that only three wavelet levels are calculated.

Data throughput can be calculated by first calculating the time required to receive, transform and then transmit a single image. Without knowledge of either the receive or transmit communication channels, it must be assumed that they are faster than the rest of the system. This means that the receive and transmit time is therefore dominated by memory speed.

The time required to receive an image is defined as the time required to write the full image into memory. The fastest way of doing this is to store the image four rows at a time, thus reducing the SDRAM access latency to the minimum possible. Equation 6.11 shows how to calculate the number of cycles required for this process (WAT refers to the Write Access Time of six cycles):

$$\frac{512}{4} \times \text{WAT} + 512 \times 512 = 128 \times 6 + 262144 = 262912 \text{ cycles} \tag{6.11}$$

The process for transmission of the image is similar. The only difference will be that the read access time (RAT) is used instead of the write access time. Making that substitution in Equation 6.11 gives a transmission time of 263296 cycles.

Transforming the rows will require a burst read operation of the row, and then one burst write to save the resulting vectors. As long as the two vectors are stored back to back they can be written using only a single SDRAM write access. The number of cycles required to transform the rows of N pixels each are given by Equation 6.12:

$$\text{RowTransformationCycles} = N \times (2N + \text{RAT} + \text{WAT}) \tag{6.12}$$

All that remains is to transform the columns. As stated in the previous section, reading and writing a column is not as straight forward as the rows. For every four pixels a new SDRAM access must be done to access a new SDRAM row. Consecutive pixels (or wavelet coefficients) in an image row are stored together as one 32bit word. This doubles the transfer rates for image row accesses, but such is not the case for consecutive pixels in the image columns. Therefore, the SDRAM controller needs to buffer all image column reads in order to send the data, at the expected rate, to the wavelet pipeline. The net result is that the latency of the wavelet pipeline and the splitter unit must now be added to the number of cycles required to read and write the image columns, in order to obtain the total number of cycles required for the transformation.

Receive:	262912			
Transmit:	263296			
Rows:	512 × 512	256 × 256	128 × 128	Total
	531968	134912	34688	701568
Columns:	512 × 512	256 × 256	128 × 128	Total
	1527808	387072	99328	2014208
Total:	3241984			

Table 6.5: *Number of Cycles Required to Transform One Image*

If the latency of the wavelet pipeline and of the splitter unit is L, the formula for calculating the total number of cycles is given by Equation 6.13:

$$\text{ColumnTransformationCycles} = N \times \left(\text{L} + \frac{N}{4} \times (\text{RAT} + \text{WAT}) + 2N\right) \tag{6.13}$$

The actual latency of the wavelet pipeline and splitter unit together is at most 40 cycles.

The total number of cycles required to transform one image is given in Table 6.5.

With a SDRAM clock frequency of 133MHz, the time required per image is $\frac{3241984}{133 \times 10^6} = 24.38$ milliseconds. That gives a throughput of 41.02 images per second or 10.26MB per second.

The calculation above does not account for SDRAM refreshing. In an SDRAM, data is stored by charging a capacitor. This charge leaks away over time. To prevent data loss, regular refresh commands must be sent to the SDRAM. However, explicit refresh commands are not required in the design under consideration, for the following reasons:

- SDRAMs guarantee data retention for 64 milliseconds. The time required to process one image is less than half that, and data retention past 64 milliseconds is not required.

- The function of a refresh command is to activate and then precharge SDRAM rows. This is also done by all read and write operations. Since the wavelet transform will read and write to all the required memory locations in a fixed pattern, all these locations are refreshed regularly.

Although this result is attractive, it is possible to improve on it. From the table it is clear that the image columns require significantly more time than the image rows to be transformed. By reducing the number of cycles required to transform the columns, it is possible to greatly improve the overall system data throughput.

The column transformations require more time than for the rows because:

1. Each group of four pixels is stored in separate SDRAM rows. In order to access a new row, a new memory read/write cycle must be initiated, requiring the full read/write access latency.

2. Consecutive pixels inside the image rows are stored together as a 32bit word. The result of this is that the image columns must be buffered by the SDRAM controller. The effect of this is that the latency of the wavelet pipeline must be added to the time required for the transformation.

Both of these problems can be addressed by using two SDRAM ICs instead of a single IC. By placing consecutive image rows at the same SDRAM row and bank addresses but in separate ICs, the second problem is completely removed because consecutive pixels in the image rows and columns can now be easily read/written together. A second IC will also add four extra SDRAM banks for each SDRAM row address, with the result that eight image column pixels can be read before a new SDRAM row must be accessed. This thereby halves the memory access time overhead.

Using a second SDRAM IC as described will increase the overall data throughput to 12.95MB per second, a 26% improvement. Adding more SDRAM ICs will further reduce the memory access time overhead for reading/writing image columns. Although it is possible to continue adding more ICs until the image column transformation time is equal to the row transformation time, this is impractical, as it will require 128 SDRAM ICs for images of 512×512 pixels.

Another possible way of improving performance is to use faster SDRAM, and thus clocking the whole system faster. Currently the fastest SDRAM can run at clock frequencies of up to 200MHz. Due to the doubling of the data rate, the FPGA system would need to run at 400MHz. However, this is right on the upper limit of current FPGA technologies. Therefore an improvement in system performance driven by increasing the clock frequency requires a move from FPGAs to application specific integrated circuits (ASICs). An ASIC implementation has two advantages:

1. FPGAs contain programmable gates. Because these gates are programmable, they have been built using much more transistors than a regular, non-programmable gate would have used. These extra transistors increase the power consumption of FPGAs and also limit the maximum clock speed. ASICs are not programmable and thus do not have these extra transistors.

2. FPGAs are based on SRAM (Static Random Access Memory) technology, which is susceptible to radiation damage or single event upsets in space. On the other hand, ASICs are not susceptible to such environmental hazards, unless they contain

memory cells. Memory cells will continue to be susceptible, but error checking and correcting circuitry can be added to memory cells to combat this weakness.

As previously stated, the analysis did not take into account the possibility that the input or output communication channels may run at a lower data rate than the maximum throughput of the wavelet system. However, it can be seen that, if any of these channels are limited, they will hinder overall system performance, and the actual performance of the wavelet unit will not matter anymore. It is therefore important to ensure that these channels are able to operate at least as fast as the systems that they service in order that the whole system can perform optimally.

6.5 Conclusion

The MSMI (Micro-Satellite Multi-sensor Imager) system will output data at a rate of 1000 Mbps. Using two SDRAM ICs, the wavelet transform unit runs at 12.95MB per second or 103 Mbps. Ten of these units, in parallel, would be required in order to ensure that the MSMI systems's productivity is not limited.

There are some methods of increasing the performance of the wavelet pipeline implementation that have not been examined in this research. It is possible to transform multiple images in parallel on parallel units. It is also possible to implement parallelism on the image row and image column level if more than one SDRAM IC is used. It would then be possible to supply two or more parallel wavelet pipelines with data streams. This strategy can potentially lead to significant processing speed increases and should be investigated in future.

Chapter 7

Implementing EZW on an Embedded Processor

In Chapter 6, a hardware implementation of the wavelet transform was discussed. In the current chapter, the implementation of the EZW algorithm on a general purpose embedded processor will be investigated. The starting point for the investigation is to establish an estimation of the execution time of the EZW algorithm.

The MSMI system generates data at a rate of 1000 Mbps, and the compression system for the images should not be permitted to limit that data rate. In this chapter we assume that images are 512×512 pixels in size, and each pixel is eight bits in size. Therefore, each image is 2 Mbits in size. The execution time per image that will not limit the productivity of the MSMI system is therefore 2 milliseconds per image.

7.1 Estimating Execution Time

The first step to calculating the worst case execution time of the algorithm is to break the algorithm down into simpler functions. This enables a profiling tool such as gprof (the GNU Profiling Tool [15]) to be used to determine the maximum number of times each function is called. The program can be run using different test images, with the maximum number of executions for each function being recorded. The usage of the GNU Profiling Tool in the current research, and the results thereof, is provided in Appendix C.

The next step is to calculate the worst case execution time (WCET) for each function. In order to do this, the source code needs to be compiled to assembler instructions. It is possible to determine the WCET for each function by calculating the WCET of each assembler instruction. Since assembler instructions and their execution times will vary from platform to platform, this method for calculating the WCET will require that a target platform be selected and then evaluated.

To calculate the WCET for the whole algorithm, the WCET of each function is multiplied by the maximum number of calls made to that function. All the products are then added together, the result of which is the WCET for the whole algorithm.

The first platform investigated was an AMD Alchemy Au1500 processor running at 400MHz.

7.1.1 Evaluation of the AMD Alchemy Au1500

The Au1500 processor [2] is based on the MIPS32 [11] instruction set. It was designed to deliver high performance at low power, typically 700mW for the 400MHz version, making it a good choice for a satellite environment, where power consumption must be kept as low as possible.

The following assumptions were made when calculating the WCET:

- Instructions are always located in the instruction cache. The Au1500 has a 16kB instruction cache, although it is expected that the total program size will be smaller than this. Thus this assumption is warranted.

- All data accesses (loads and stores) result in cache misses. Because the SDRAM clock runs four times slower than the core clock in the Au1500, accessing data is a slow process. Therefore, by assuming that all data accesses result in cache misses, the absolute worst case execution time can be calculated.

- The Au1500 uses a FIFO buffer to write data to the SDRAM if the address (where the data should be written to) was not cached at the time the store instruction was executed. To simplify calculations, it is assumed that there will always be space in the write buffer, and that data won't be read until after it has been written to the external RAM.

- During the first analysis of the assembler instructions, load and store instructions were treated as one group. Therefore the exact ratio of loads versus stores is not known. It is assumed that the ratio is one to one, and in the case of an uneven number of load/store instructions, it is assumed that there is one extra load instruction.

- It is assumed that the SDRAM used has a CAS latency of only two cycles at a clock frequency of 100MHz.

- It is assumed that the SDRAM will not be accessed during refresh cycles, or while any other operations are in progress, which would add unpredictable delays.

Using the above assumptions, and with reference to the information in the Au1500's data book, the execution times of the individual instructions are as follows:

- Load instructions require 49 cycles to complete.
- Stores will only require one cycle to write to the write buffer.
- "MUL"-instructions require two cycles to perform a 32×32-bit multiplication, and an additional cycle to store the result in the target register. Therefore a total of three cycles are allocated to each multiplication.
- Divide instructions may require up to 35 cycles to complete. Adding one cycle to store the result brings the total to 36 cycles.
- Branch and jump instructions requires two cycles when the jump is made. It is assumed that all branches will be taken.
- No coprocessor-, control- or any other special instructions are used.

The procedure for calculating the WCET is as follows:

1. Using the WCET for each assembly instruction and the assembly implementation of each function, calculate the WCET of each function.
2. Multiply the WCET of each function by the worst case number of times that function is called (from profiling information).
3. Add all the total WCETs of the functions to get the WCET for the whole algorithm.

The WCET for the algorithm was calculated to be 10.75 seconds. Using the same method, but assuming that all data load instructions will result in cache hits (ie. all load instructions require a single cycle to execute), the execution time was calculated to be 1.077 seconds. This is not the best case execution time, but it provides an example of how the data cache can influence the actual execution time. These results were obtained using a variety of test images with seven wavelet scales and a compression ratio of $8.192 : 1$. The influence of wavelet scales and the compression ratio will be discussed in Section 7.3.

7.2 Cache Considerations

The two calculated WCETs differ by one order of magnitude. This difference is caused only by the effectiveness of the data cache. It is this substantial difference that makes it important to be able to predict the cache performance to some degree.

Predicting the behaviour of a cache is a complex task, but some processors do provide cache-instructions which can be used to both increase performance and make the cache behaviour more predictable. The MIPS instruction set includes (amongst other cache instructions) the prefetch (PREF) instruction. This instruction is used to load data from the external memory into the cache before it is actually required by the application. By using this and other cache instructions, the cache can be carefully managed by the software in order to ensure optimal cache performance. If implemented for the EZW algorithm, it is possible to maintain a very high cache hit ratio. A perfect hit ratio cannot be guaranteed because the prefetch instruction is only advisory - ie. the cache may not necessarily perform the prefetch.

7.3 Image Quality vs Execution Time

The image quality is affected by the following two parameters: the number of wavelet scales; and the compression ratio.

7.3.1 Wavelet Scales

Figure 7.1 shows the influence that the number of wavelet scales has on the image quality. For the test image used, the PSNR and mean absolute pixel error graphs show that the best image quality is achieved by using three wavelet scales. The maximum absolute pixel error graph shows the best quality to be at four wavelet scales for this particular image. The position of these best quality points will move when the image and compression ratios are changed, but the graphs will always have the same general shapes.

The number of scales also has an influence on the execution time. Table 7.1 shows the approximate worst case execution times for different numbers of wavelet scales. These results were obtained with only one image and for only the 400MHz AMD Alchemy processor.

The EZW algorithm exploits the zerotree structure (see Section 4.2.2). When only one or two wavelet scales are used, there are not many zerotree structures in the image, and the EZW algorithm becomes less effective. Without zerotree structures, the execution time increases, since the EZW algorithm must analyse significantly more coefficients per coding pass than it would were there more zerotree structures.

For the MIPS instruction set, the lowest execution time for the EZW algorithm is in the range of three to six wavelet scales. The more scales used in the transform, the longer the wavelet transform will take to complete. Therefore, for these test parameters, the ideal number of scales will be either three or four.

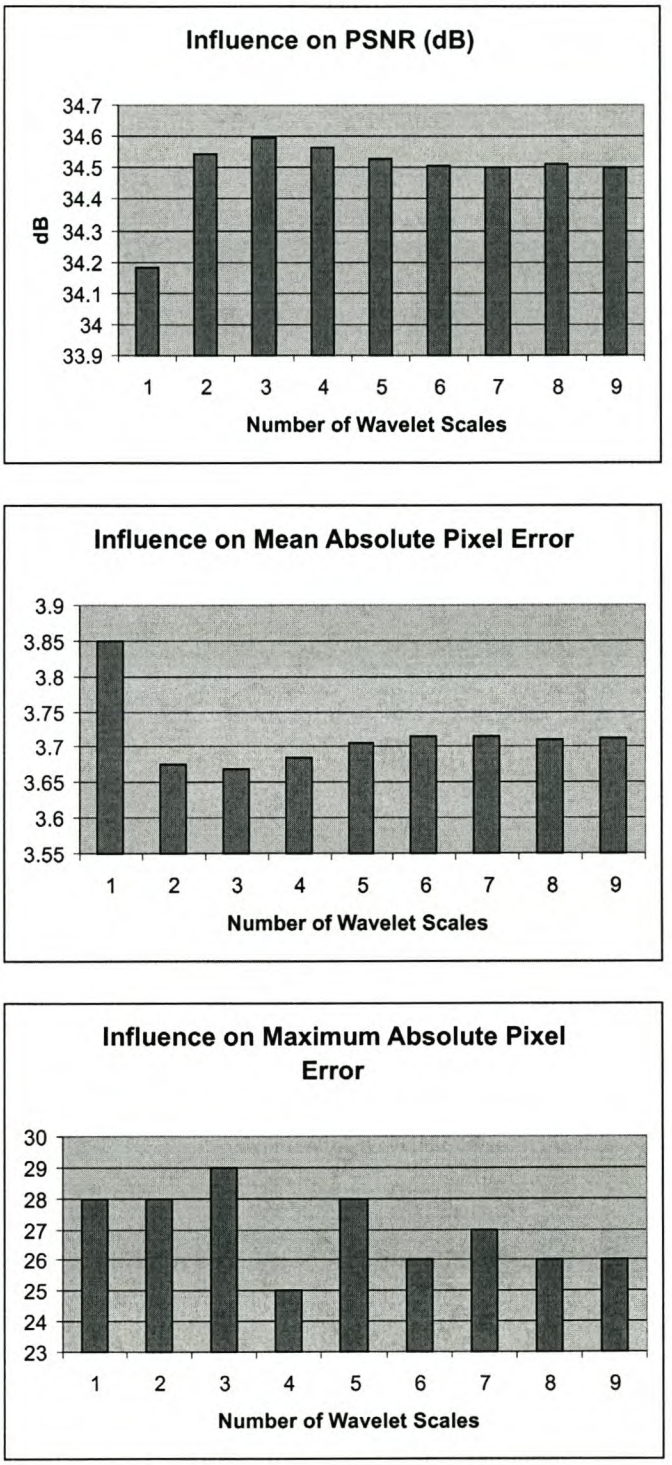

Figure 7.1: *Influence of the number of wavelet scales on image quality*

Number of Scales	WCET (seconds)
1	18.11
2	10.36
3	8.159
4	7.670
5	8.202
6	8.176
7	8.813
8	8.810
9	13.30

Table 7.1: *Number of scales' influence on WCET*

7.3.2 Compression Ratio

The compression ratio has a greater influence on execution time than the number of wavelet scales. The lower the compression ratio (and higher the image quality), the more passes must be made to fill the byte budget, and the longer it will take the EZW algorithm to complete. Figure 7.2 shows how the compression ratio influences the WCET when all other test parameters are kept constant.

From Figure 7.2 it can be seen that the execution time increases by fifteen percent when the compression ratio is reduced from 0.25 to 0.5 bits per pixel. The graph is an exponential function, and execution time increases rapidly as the compression ratio is reduced more. Therefore, the execution time increases significantly at low compression ratios. However, it is at these low compression ratios that the image quality is best.

7.4 Conclusion

These results show that a MIPS-core running at only 400MHz will not be able to meet the timing requirements. A significantly higher speed processor, perhaps with a DDR-SDRAM memory interface to increase the memory bandwidth, will be needed. A new processor that recently became available is Intrinsity's FastMIPS processor, utilising a MIPS−32 core running at 2GHz with a large on-chip cache and a high speed DDR SDRAM controller built-in. This processor will be capable of approaching the performance requirement more closely, and an analysis of the WCET of the EZW algorithm on this processor should be made.

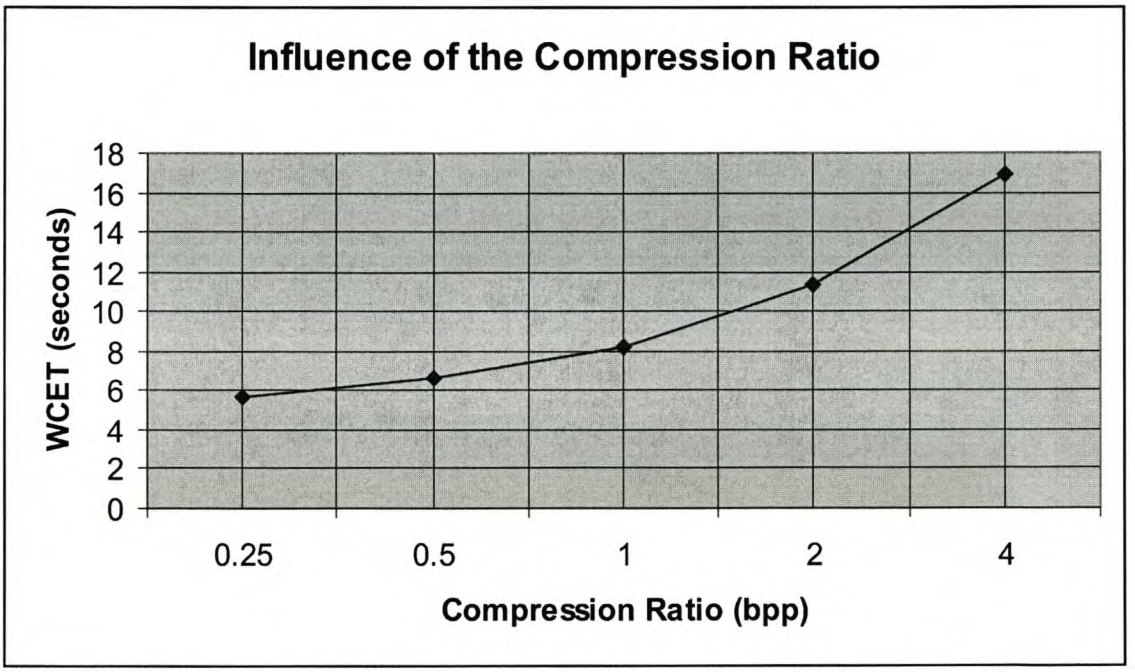

Figure 7.2: *Influence of the Compression Ratio on the WCET*

Chapter 8

Comparisons with Commercial Systems

In this section, the results of the EZW implementation investigated in this thesis will be compared with those of IMEC (Interuniversity MicroElectronics Center) and ESA's (European Space Agency) FlexWave-II image compression system, and with those of Amphion's CS6510 JPEG2000 Encoder IC.

8.1 The FlexWave-II System

The FlexWave-I system was developed by IMEC as a wavelet-based still image compressor. The main purpose for the implementation was remote sensing applications for earth observation satellites. The FlexWave-II system is based on the FlexWave-I, but utilises some new algorithms (like a local wavelet transform) to increase the performance, and to add support for push-broom images and spatial scalability.

Xilinx Virtex2000E-08 FPGA	
Operating Frequency:	41MHz
Maximum Throughput:	10MB per second
0.18μm-technology ASIC	
Operating Frequency:	80MHz (estimated)
Maximum Throughput:	20MB per second (estimated)

Table 8.1: *Data Throughput of the FlexWave-II System [7]*

Table 8.1 shows the data throughput of the FlexWave-II system. The data throughput results are for the whole FlexWave-II system and includes its wavelet transform, its embedded zerotree encoder and its arithmetic encoder.

In Chapter 6 the wavelet transform implementation was calculated to be able to sustain a data throughput of 12.95MB per second. This data throughput is almost 30% faster than the FPGA implementation of the FlexWave-II system, but runs at a significantly higher operating frequency. A design running at a lower operating frequency will have the following advantages:

- Power consumption is proportionate to the switching frequency. Thus a lower operating frequency will lead to lower power consumption.
- Slower and cheaper devices can be used. These devices are also typically made with larger silicon processes, with the result that they are less susceptible to single event upsets.
- Electromagnetic interference is a problem at higher frequencies. Therefore this problem would be reduced at lower operating frequencies.

Table 8.2 shows the image quality results for the FlexWave-II and the EZW implementation. The same test image, an image of a weather system over the North-Atlantic ocean, was used in both tests.

With the PSNR quality measure, the FlexWave-II system has much better image quality than the EZW implementation. However, the absolute pixel error measure gives a better idea of the quality of the scientific data held in image. In this area, the implementation of the EZW algorithm is ahead by a significant margin.

Note that these results are for a single test image. To make a better comparison, the results of a whole range of images should be compared.

8.2 Amphion CS6510 JPEG2000 Encoder

The CS6510 JPEG2000 Encoder [5] is a high performance application specific device. The core is fully compliant with the ISO/IEC 15444-1 JPEG2000 Image Coding System Standard.

The performance of the CS6510 is reported to be 20 mega-pixels per second (or 250 millisecond compression time for a five mega-pixel still image). With eight bit grey-scale images this implies a data throughput of 20MB. This performance is for a 0.18μm-technology ASIC implementation running at 180MHz. The performance is in the same class as the FlexWave-II system.

The JasPer [1] Software tool was used to compare the image quality of JPEG2000 with the EZW image compression system. Both the JasPer and the CS6510-core are

Results of FlexWave-II System [17]			
Compression Ratio (bpp)	PSNR (dB)	Maximum Absolute Pixel Error	Mean Absolute Pixel Error
0.25	37.17	124	9.91
0.5	40.84	71	6.72
1.0	45.60	39	4.00
2.0	51.49	17	2.05

Results of the EZW Implementation						
Compression Ratio	0.25 bpp			0.5 bpp		
Number of Wavelet Scales	3	5	7	3	5	7
PSNR (dB)	36.312	36.197	36.200	37.815	37.877	37.867
Mean Absolute Pixel Error	3.02	3.07	3.07	2.58	2.56	2.57
Max Absolute Pixel Error	25	26	26	16	16	16
Compression Ratio	1.0 bpp			2.0 bpp		
Number of Wavelet Scales	3	5	7	3	5	7
PSNR (dB)	40.161	40.020	39.991	42.720	42.435	42.355
Mean Absolute Pixel Error	1.96	2.00	2.00	1.45	1.50	1.52
Max Absolute Pixel Error	13	12	13	10	10	12

Table 8.2: *Image Quality Comparison between the FlexWave-II System and EZW*

fully compliant with the ISO/IEC 15444-1 JPEG2000 standard, and therefore the JasPer image quality results can be used as reference.

The following series of figures show comparisons for four test images. Figure 8.1 compares the PSNR results. The maximum and mean absolute pixel errors are compared in Figures 8.2 and 8.3 respectively.

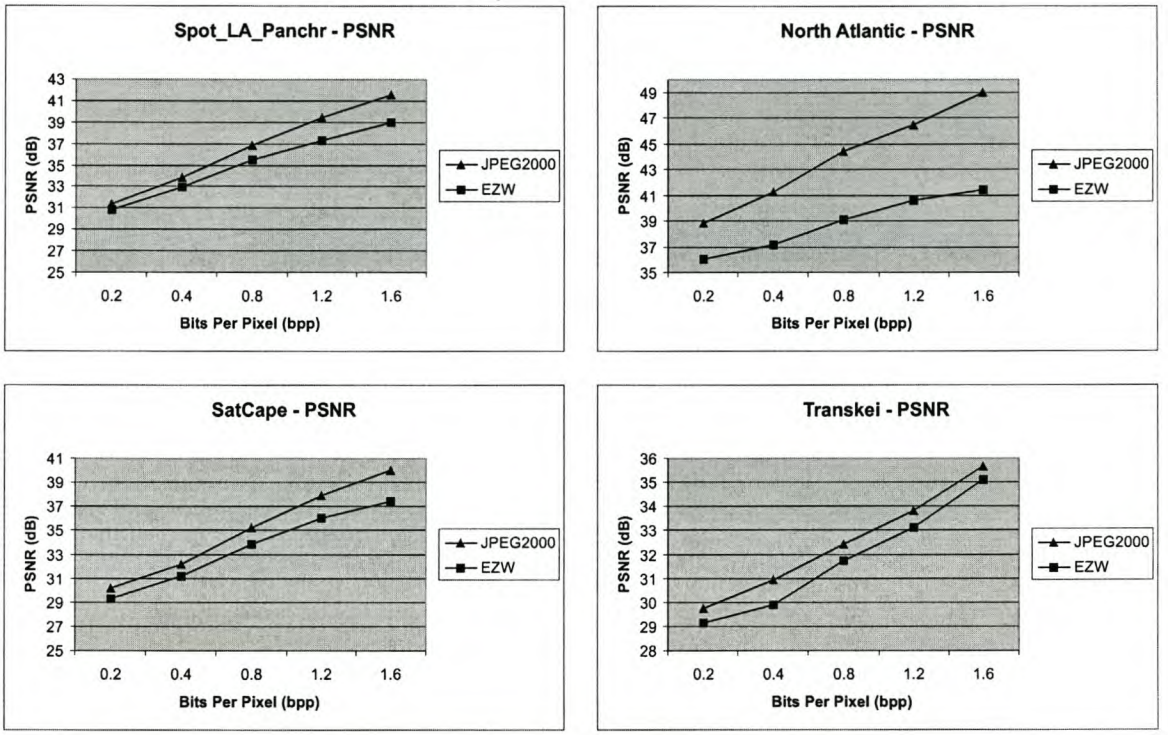

Figure 8.1: *PSNR Comparison between JPEG2000 and EZW*

On average, the JPEG2000-algorithm's results are the better of the two. With regards to the maximum absolute pixel error the two systems are closely matched. However, the EZW algorithm is 2.23dB behind JPEG2000 with the PSNR, and 0.713 behind with the mean absolute pixel error.

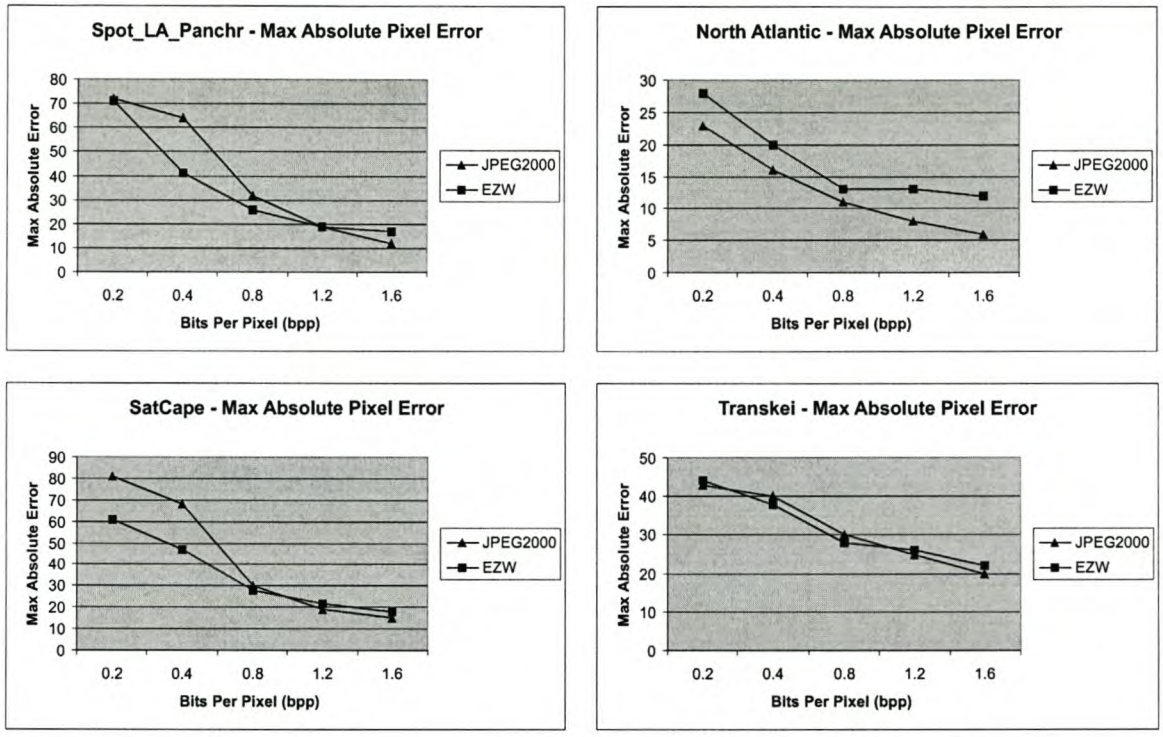

Figure 8.2: *Maximum Absolute Pixel Error Comparison between JPEG2000 and EZW*

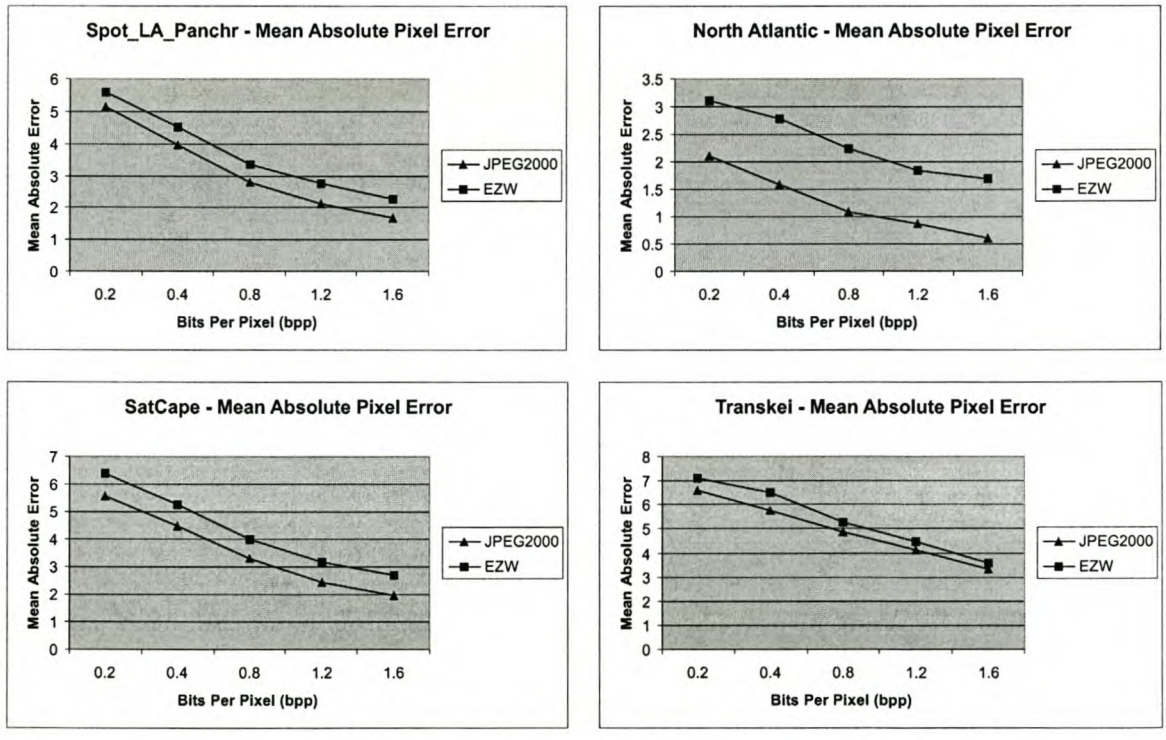

Figure 8.3: *Mean Absolute Pixel Error Comparison between JPEG2000 and EZW*

Chapter 9

Conclusions and Recommendations

This chapter will draw some conclusions from the described implementation of the wavelet transform and the EZW algorithm. Image quality, data throughput performance and power consumption will be discussed. Finally the chapter will also present some recommendations for future work.

9.1 Conclusions

All image quality tests showed that JPEG2000 is superior to EZW. The FlexWave-II system also received superior results in the PSNR image quality tests, but the reported mean and maximum absolute pixel errors are much bigger for the FlexWave-II than for EZW.

The wavelet transform implementation is in the same data throughput class as the FlexWave-II and the CS6510 systems. Both the FlexWave-II system and the wavelet implementation of this thesis can sustain a data throughput of about 10 MB per second in FPGAs. The CS6510 reports 20 MB per second in an ASIC implementation, which is the same as that reported for the ASIC implementation of the FlexWave-II system. However, it should be noted that the FlexWave-II system runs at lower operating frequencies than our wavelet transform implementation.

The execution time of the EZW algorithm on a general purpose processor is too high, and does not attain in the goal of the project. Other implementations of the EZW algorithm, or a different algorithm, should therefore be investigated.

A high-performance processor implementation of the EZW algorithm was suggested in Chapter 7. These processors typically consume more than ten watt during operation. Power consumption greater than ten watt is not ideal for micro-satellites, which have limited power budgets.

Altera Quartus software reported the power consumption of the wavelet pipeline to

be 417mW. The power consumption will scale as the extra components required by the described wavelet transform unit are added.

9.2 Recommendations

Recommendations for future work include:

- **Image Size:** The current implementation only takes into account images where the width and height are equal and a power of two (for example, 512×512 pixels). These image size restrictions allowed performance advantages. Satellites with push-broom imaging sensors will need to temporarily store the images produced by the sensors, before passing them to the image compression system.

- **HDL Implementation of the EZW algorithm:** The FlexWave-II system utilises a quantiser similar to the EZW algorithm, which has been implemented in hardware. If the EZW algorithm can also be implemented in hardware, it is possible to achieve better performance without the high power consumption that is typical of high-performance processors.

- **Alternate Monitoring Function for the EZW Algorithm:** Currently the EZW algorithm uses a function that monitors the size of the compressed data, in order to decide when to stop. What is needed is some means of guaranteeing the absolute pixel errors. If a function can be implemented in the EZW algorithm that can keep track of the absolute pixel errors while compressing, it will be possible to end the compression when a specific quality level has been reached. With such a monitoring function, it will also be possible to predict the impact that the lossy compression will have on the information extraction processes. Research into this topic is currently in progress at the Vrije Universiteit Brussel [3].

- **Effects of Noise on the EZW Data Stream:** No study had been made of how bit errors will affect the EZW decoder if the communication channel between the EZW encoder and the decoder is noisy. Since a perfect channel can never be guaranteed, it is important to understand what impact noise will have on the decoder.

Bibliography

[1] ADAMS, M. D., "The JasPer Project." http://www.ece.uvic.ca/ mdadams/jasper/. August 2001.

[2] ADVANCED MICRO DEVICES, INC., Sunnyvale, CA. *AMD Alchemy Solutions Au1500 Processor Data Book*, 2003.

[3] ALECU, A. *et al.*, "Wavelet-based fixed and embedded L-infinite-constrained image coding." Tech. Rep. IRIS-TR-0087, Vrije Universiteit Brussel, Dept. ETRO, July 2002.

[4] ALTERA CORPORATION, San Jose, CA. *Altera Stratix FPGA Data Sheet*, 2002.

[5] AMPHION, Belfast, Northern Ireland, UK. *CS6510 JPEG2000 Encoder*, 2002.

[6] DAUBECHIES, I. and SWELDENS, W., "Factoring Wavelet Transforms into Lifting Steps." *J. Fourier Anal. Appl.*, 1998, Vol. 4, No. 3, No. 3, pp. 245–267.

[7] IMEC, "IMEC FlexWave." http://www.imec.be/design/multimedia/flexwave.shtml. December 2002.

[8] MARTIN, G., "Range encoding: an algorithm for removing redundancy from a digitised message." *Video and Data Recording Conference*, March 1979.

[9] MICRON TECHNOLOGY, INC., Boise, ID. *Micron MT48LC2M32B2 64Mbit SDRAM Data Sheet*, 2002.

[10] MIPS, "MIPS Technologies Inc." http://www.mips.com/. September 2003.

[11] MIPS TECHNOLOGIES, Mountain View, CA. *MIPS32 Architecture For Programmers Volume III: The MIPS32 Privileged Resource Architecture*, 2001.

[12] SCHINDLER, M., "The Range Coder." http://www.compressconsult.com/rangecoder/. October 1999.

[13] SHAPIRO, J. M., "Embedded Image Coding Using Zerotrees of Wavelet Coefficients." *IEEE Transactions on Signal Processing*, December 1993, Vol. 41, No. 12, pp. 3445–3462.

[14] SWELDENS, W., "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions." in *Wavelet Applications in Signal and Image Processing III* (LAINE, A. F. and UNSER, M. (Eds)), pp. 68–79, Proc. SPIE 2569, 1995.

[15] THE GCC TEAM, "GCC Home Page - GNU Projects - Free Software Foundation." `http://www.gnu.org/software/gcc/gcc.html`. August 2003.

[16] VERMOTE, E. *et al.*, "Second Simulation of the Satellite Signal in the Solar Spectrum, 6S: An overview." *IEEE Trans. Geosc. and Remote Sens.*, 1997, Vol. 35, No. 3, No. 3, pp. 675–686.

[17] WATZEELS, E., "Requirement analysis of image data compression systems in space missions." Master's thesis, Katholieke Hogeschool Brugge-Oostende, 2002.

Appendix A

VHDL Source of Wavelet Implementation

VHDL source code for the wavelet pipeline and supporting units are given below. Some units were implemented using Altera's Megafunctions. For these units some details of the setup of the Megafunction is included, but not the source VHDL as this can be recreated with the Quartus II software.

Details on the design, including block diagrams, can be found in Chapter 6.

A.1 Alpha Stage D-Pipe

```
-- Eduard Kriegler
-- June/July 2003
-- AlphaDPipe.vhd
-- Implements a small delay/fifo buffer

library ieee;
use ieee.std_logic_1164.all;

entity AlphaDPipe is
    port (
        clk     : in std_logic;
        din     : in std_logic_vector(15 downto 0);
        dout    : out std_logic_vector(15 downto 0)
    );
end entity AlphaDPipe;
```

```
architecture rtl of AlphaDPipe is

signal lev1, lev2, lev3, lev4 : std_logic_vector(15 downto 0);

begin
    process (clk) is
    begin
        if rising_edge(clk) then
            dout <= lev4;
            lev4 <= lev3;
            lev3 <= lev2;
            lev2 <= lev1;
            lev1 <= din;
        end if;
    end process;
end architecture rtl;
```

A.2 Alpha Stage S-Pipe

```
-- Eduard Kriegler
-- June/July 2003
-- AlphaSPipe.vhd
-- Implements a small delay/fifo buffer

library ieee;
use ieee.std_logic_1164.all;

entity AlphaSPipe is
    port (
        clk     : in std_logic;
        sin     : in std_logic_vector(15 downto 0);
        sout    : out std_logic_vector(15 downto 0)
    );
end entity AlphaSPipe;

architecture rtl of AlphaSPipe is
```

```
signal lev1, lev2, lev3, lev4, lev5, lev6, lev7, ...
   lev8, lev9, lev10 : std_logic_vector(15 downto 0);

begin
    process (clk) is
    begin
        if rising_edge(clk) then
            sout  <= lev10;
            lev10 <= lev9;
            lev9  <= lev8;
            lev8  <= lev7;
            lev7  <= lev6;
            lev6  <= lev5;
            lev5  <= lev4;
            lev4  <= lev3;
            lev3  <= lev2;
            lev2  <= lev1;
            lev1  <= sin;
        end if;
    end process;
end architecture rtl;
```

A.3 Beta Stage D-Pipe

```
-- Eduard Kriegler
-- June/July 2003
-- BetaDPipe.vhd
-- Implements a small delay/fifo buffer

library ieee;
use ieee.std_logic_1164.all;

entity BetaDPipe is
    port (
        clk     : in std_logic;
        din     : in std_logic_vector(15 downto 0);
        dout    : out std_logic_vector(15 downto 0)
```

```
    );
end entity BetaDPipe;

architecture rtl of BetaDPipe is

signal lev1, lev2, lev3, lev4, lev5, lev6, lev7, ...
   lev8, lev9, lev10 : std_logic_vector(15 downto 0);

begin
    process (clk) is
    begin
        if rising_edge(clk) then
            dout  <= lev10;
            lev10 <= lev9;
            lev9  <= lev8;
            lev8  <= lev7;
            lev7  <= lev6;
            lev6  <= lev5;
            lev5  <= lev4;
            lev4  <= lev3;
            lev3  <= lev2;
            lev2  <= lev1;
            lev1  <= din;
        end if;
    end process;
end architecture rtl;
```

A.4 Gamma Stage S-Pipe

```
-- Eduard Kriegler
-- June/July 2003
-- GammaSPipe.vhd
-- Implements a small delay/fifo buffer

library ieee;
use ieee.std_logic_1164.all;
```

```
entity GammaSPipe is
    port (
        clk     : in std_logic;
        sin     : in std_logic_vector(15 downto 0);
        sout    : out std_logic_vector(15 downto 0)
    );
end entity GammaSPipe;

architecture rtl of GammaSPipe is

signal lev1, lev2, lev3, lev4, lev5, lev6, lev7, ...
   lev8, lev9, lev10 : std_logic_vector(15 downto 0);

begin
    process (clk) is
    begin
        if rising_edge(clk) then
            sout <= lev10;
            lev10<= lev9;
            lev9 <= lev8;
            lev8 <= lev7;
            lev7 <= lev6;
            lev6 <= lev5;
            lev5 <= lev4;
            lev4 <= lev3;
            lev3 <= lev2;
            lev2 <= lev1;
            lev1 <= sin;
        end if;
    end process;
end architecture rtl;
```

A.5 Delta Stage D-Pipe

```
-- Eduard Kriegler
-- June/July 2003
-- DeltaDPipe.vhd
```

```
-- Implements a small delay/fifo buffer

library ieee;
use ieee.std_logic_1164.all;

entity DeltaDPipe is
    port (
        clk     : in std_logic;
        din     : in std_logic_vector(15 downto 0);
        dout    : out std_logic_vector(15 downto 0)
    );
end entity DeltaDPipe;

architecture rtl of DeltaDPipe is

signal lev1, lev2, lev3, lev4 : std_logic_vector(15 downto 0);

begin
    process (clk) is
    begin
        if rising_edge(clk) then
            dout <= lev4;
            lev4 <= lev3;
            lev3 <= lev2;
            lev2 <= lev1;
            lev1 <= din;
        end if;
    end process;
end architecture rtl;
```

A.6 Duplicator

```
-- Eduard Kriegler
-- July 2003
-- Duplicator
-- Used to fix wavelet data streams for boundary
-- conditions
```

```
library ieee;
use ieee.std_logic_1164.all;

entity Dublicator is
    port (
        clk     : in std_logic;
        control : in std_logic;
        din     : in std_logic_vector(15 downto 0);
        dout    : out std_logic_vector(15 downto 0)
    );
end entity Dublicator;

architecture rtl of Dublicator is

signal output : std_logic_vector(15 downto 0);

begin

    with control select
        dout <= din when '0',
                output when '1',
                (others => 'X') when others;

    process (clk) is
    begin
        if rising_edge(clk) then
            output <= din;
        end if;
    end process;
end architecture rtl;
```

A.7 Edge Generator

```
-- Eduard Kriegler
-- July 2003
```

```
-- Edge Generator
-- Fixes the wavelet data streams for boundary conditions

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity EdgeGen is
    port (
        iStream : in std_logic_vector(15 downto 0);
        clk     : in std_logic;
        trigger : in std_logic;
        oStream : out std_logic_vector(15 downto 0)
    );
end entity EdgeGen;

architecture rtl of EdgeGen is

signal latch1, latch0 : std_logic_vector(15 downto 0);

begin
    process (clk) is
    begin
        if rising_edge(clk) then
            latch1 <= latch0;
            latch0 <= iStream;
            if trigger = '0' then
                oStream <= iStream;
            else
                oStream <= latch0 - latch1;
            end if;
        end if;
    end process;
end architecture rtl;
```

A.8 Multipliers

The multipliers are implemented with Altera's Megafunctions. Each multiplier is a signed 16×16-bit multiplier with a 32bit output. Port B input is constant and set to the appropriate multiplication constant (see Table 6.1).

A.9 Arithmetic Right Shifter

This unit is implemented with Altera's "LPM_CLSHIFT"-Megafunction. The data width is 32bits and the distance width is 5bits.

A.10 Signed Adder

```
-- Eduard Kriegler
-- June 2003
-- SignedAdd.vhd
-- Performs a Signed Add on the input words

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity SignedAdd is
port (
A, B : in std_logic_vector(15 downto 0);
result : out std_logic_vector(15 downto 0)
);
end entity SignedAdd;

architecture rtl of SignedAdd is
begin
result <= A + B;
end architecture rtl;
```

A.11 Wait Adder

```
-- Eduard Kriegler
```

```
-- June/July 2003
-- WaitAdd.vhd
-- Adds consecutive input words

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity WaitAdd is
    port (
        clk           : in std_logic;
        operands      : in std_logic_vector(15 downto 0);
        result        : out std_logic_vector(15 downto 0)
    );
end entity WaitAdd;

architecture rtl of WaitAdd is
begin
    process (clk) is
    variable reg : std_logic_vector(15 downto 0);
    begin
        if rising_edge(clk) then
            result <= reg + operands;
            reg := operands;
        end if;
    end process;
end architecture rtl;
```

A.12 Wavelet Pipeline

```
-- Eduard Kriegler
-- June/July 2003
-- Wavelet Transform Pipeline
-- Currently implementing a lifting Bior (9,7) filter

library ieee;
use ieee.std_logic_1164.all;
```

```
entity WaveletPipe is
    port (
        clk         : in std_logic;
        aclr        : in std_logic;
        dub0Ctrl,
        dub1Ctrl    : in std_logic;
        eGen0Ctrl,
        eGen1Ctrl   : in std_logic;
        s, d        : in std_logic_vector(15 downto 0);
        sf, df      : out std_logic_vector(15 downto 0)
    );
end entity WaveletPipe;

architecture rtl of WaveletPipe is

component WaitAdd is
    port (
        clk         : in std_logic;
        operands    : in std_logic_vector(15 downto 0);
        result      : out std_logic_vector(15 downto 0)
    );
end component WaitAdd;

component Mul_Alpha is
    port (
        dataa       : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
end component Mul_Alpha;

component Mul_Beta is
    port (
        dataa       : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
end component Mul_Beta;
```

```
component Mul_Gamma is
    port (
        dataa        : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
end component Mul_Gamma;

component Mul_Delta is
    port (
        dataa        : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
end component Mul_Delta;

component SAR is
    port (
        distance     : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        data         : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
end component SAR;

component SignedAdd is
    port (
        A, B         : in std_logic_vector(15 downto 0);
        result       : out std_logic_vector(15 downto 0)
    );
end component SignedAdd;

component AlphaDPipe is
    port (
        clk      : in std_logic;
        din      : in std_logic_vector(15 downto 0);
        dout     : out std_logic_vector(15 downto 0)
    );
end component AlphaDPipe;
```

```
component AlphaSPipe is
    port (
        clk     : in std_logic;
        sin     : in std_logic_vector(15 downto 0);
        sout    : out std_logic_vector(15 downto 0)
    );
end component AlphaSPipe;

component Dublicator is
    port (
        clk     : in std_logic;
        control : in std_logic;
        din     : in std_logic_vector(15 downto 0);
        dout    : out std_logic_vector(15 downto 0)
    );
end component Dublicator;

component EdgeGen is
    port (
        iStream : in std_logic_vector(15 downto 0);
        clk     : in std_logic;
        trigger : in std_logic;
        oStream : out std_logic_vector(15 downto 0)
    );
end component EdgeGen;

component BetaDPipe is
    port (
        clk         : in std_logic;
        din         : in std_logic_vector(15 downto 0);
        dout        : out std_logic_vector(15 downto 0)
    );
end component BetaDPipe;

component GammaSPipe is
    port (
```

```
        clk         : in std_logic;
        sin         : in std_logic_vector(15 downto 0);
        sout        : out std_logic_vector(15 downto 0)
    );
end component GammaSPipe;

component DeltaDPipe is
    port (
        clk         : in std_logic;
        din         : in std_logic_vector(15 downto 0);
        dout        : out std_logic_vector(15 downto 0)
    );
end component DeltaDPipe;

-- Pre-Alpha Signals:
signal s0           : std_logic_vector(15 downto 0);

-- Alpha Phase Signals:
signal WaitAdd0_out : std_logic_vector(15 downto 0);
signal MulAlphaIn   : std_logic_vector(15 downto 0);
signal MulAlphaOut  : std_logic_vector(31 downto 0);
signal SAR0In       : std_logic_vector(31 downto 0);
signal SAR0Out      : std_logic_vector(31 downto 0);
signal Add0I1       : std_logic_vector(15 downto 0);
signal Add0I2       : std_logic_vector(15 downto 0);
signal Add0Out      : std_logic_vector(15 downto 0);
signal d1, d2, d3   : std_logic_vector(15 downto 0);
signal sbeta        : std_logic_vector(15 downto 0);
signal DPipeOut     : std_logic_vector(15 downto 0);
signal d1stream     : std_logic_vector(15 downto 0);

-- Beta Phase Signals:
signal WaitAdd1_out : std_logic_vector(15 downto 0);
signal MulBetaIn    : std_logic_vector(15 downto 0);
signal MulBetaOut   : std_logic_vector(31 downto 0);
signal SAR1In       : std_logic_vector(31 downto 0);
signal SAR1Out      : std_logic_vector(31 downto 0);
```

```
signal Add1I1       : std_logic_vector(15 downto 0);
signal Add1I2       : std_logic_vector(15 downto 0);
signal Add1Out      : std_logic_vector(15 downto 0);
signal s1, s2       : std_logic_vector(15 downto 0);
signal s1stream     : std_logic_vector(15 downto 0);

-- Gamma Phase Signals:
signal WaitAdd2_out : std_logic_vector(15 downto 0);
signal MulGammaIn   : std_logic_vector(15 downto 0);
signal MulGammaOut  : std_logic_vector(31 downto 0);
signal SAR2In       : std_logic_vector(31 downto 0);
signal SAR2Out      : std_logic_vector(31 downto 0);
signal Add2I1       : std_logic_vector(15 downto 0);
signal Add2I2       : std_logic_vector(15 downto 0);
signal Add2Out      : std_logic_vector(15 downto 0);
signal GDPipeOut    : std_logic_vector(15 downto 0);
signal d2s, d2st    : std_logic_vector(15 downto 0);
signal d2stream     : std_logic_vector(15 downto 0);
signal s3           : std_logic_vector(15 downto 0);

-- Delta Phase Signals:
signal WaitAdd3_out  : std_logic_vector(15 downto 0);
signal MulDeltaIn    : std_logic_vector(15 downto 0);
signal MulDeltaOut   : std_logic_vector(31 downto 0);
signal SAR3In        : std_logic_vector(31 downto 0);
signal SAR3Out       : std_logic_vector(31 downto 0);
signal Add3I1        : std_logic_vector(15 downto 0);
signal Add3I2        : std_logic_vector(15 downto 0);
signal Add3Out       : std_logic_vector(15 downto 0);
signal d2final       : std_logic_vector(15 downto 0);
signal s2stream      : std_logic_vector(15 downto 0);

begin
    -- Pre-Alpha Stage:
    eGenPre:    EdgeGen port map (s, clk, eGen1Ctrl, s0);

    -- Alpha Stage:
```

```
WaitAdd0:    WaitAdd port map (clk, s0, WaitAdd0_out);
Mul_Alpha0:  Mul_Alpha port map (MulAlphaIn, MulAlphaOut);
SAR0:        SAR port map ("01010", SAR0In, SAR0Out);
Add0:        SignedAdd port map (Add0I1, Add0I2, Add0Out);
DPipe0:      AlphaDPipe port map (clk, d, DPipeOut);

-- Synchronise S0 and D1 streams after alpha stage:
SPipe0:      AlphaSPipe port map (clk, s0, sbeta);
Dub0:        Dublicator port map (clk, dub0Ctrl, d1, d2);

-- Beta Stage:
WaitAdd1:    WaitAdd port map (clk, d1stream, WaitAdd1_out);
Mul_Beta0:   Mul_Beta port map (MulBetaIn, MulBetaOut);
SAR1:        SAR port map ("01100", SAR1In, SAR1Out);
Add1:        SignedAdd port map (Add1I1, Add1I2, Add1Out);

-- Synchronise S1 and D1 streams after beta stage:
eGen0:       EdgeGen port map (s1, clk, eGen0Ctrl, s2);
DPipe1:      BetaDPipe port map (clk, d1stream, d3);

-- Gamma Stage:
WaitAdd2:    WaitAdd port map (clk, s1stream, WaitAdd2_out);
Mul_Gamma0:  Mul_Gamma port map (MulGammaIn, MulGammaOut);
SAR2:        SAR port map ("01010", SAR2In, SAR2Out);
Add2:        SignedAdd port map (Add2I1, Add2I2, Add2Out);

-- Synchronise S1 and D2 streams after gamma stage:
Dub1:        Dublicator port map (clk, dub1Ctrl, d2s, d2st);
SPipe1:      GammaSPipe port map (clk, s1stream, s3);

-- Delta Stage:
WaitAdd3:    WaitAdd port map (clk, d2stream, WaitAdd3_out);
Mul_Delta0:  Mul_Delta port map (MulDeltaIn, MulDeltaOut);
SAR3:        SAR port map ("01011", SAR3In, SAR3Out);
Add3:        SignedAdd port map (Add3I1, Add3I2, Add3Out);

-- Final Synchronisation:
```

```
DPipe2:        DeltaDPipe port map (clk, d2stream, d2final);

-- Latching (alpha stage and sync after alpha stage):
process (clk, aclr) is
begin
    if aclr = '1' then
        MulAlphaIn <= (others => '0');
        SAR0In     <= (others => '0');
        Add0I1     <= (others => '0');
        Add0I2     <= (others => '0');
        d1         <= (others => '0');
        d1stream   <= (others => '0');
    else
        if rising_edge(clk) then
            MulAlphaIn <= WaitAdd0_out;
            SAR0In     <= MulAlphaOut;
            Add0I1     <= SAR0Out(15 downto 0);
            Add0I2     <= DPipeOut;
            d1         <= Add0Out;
            d1stream   <= d2;
        end if;
    end if;
end process;

-- Latching (beta stage and sync after beta stage):
process (clk, aclr) is
begin
    if aclr = '1' then
        MulBetaIn    <= (others => '0');
        SAR1In       <= (others => '0');
        Add1I1       <= (others => '0');
        Add1I2       <= (others => '0');
    else
        if rising_edge(clk) then
            MulBetaIn     <= WaitAdd1_out;
            SAR1In        <= MulBetaOut;
            Add1I1        <= SAR1Out(15 downto 0);
```

```
            Add1I2        <= sbeta;
            s1            <= Add1Out;
            s1stream      <= s2;
        end if;
    end if;
end process;

-- Latching (Gamma stage and sync after Gamma stage):
process (clk, aclr) is
begin
    if aclr = '1' then
        MulGammaIn <= (others => '0');
        SAR2In     <= (others => '0');
        Add2I1     <= (others => '0');
        Add2I2     <= (others => '0');
        d2s        <= (others => '0');
        d2stream   <= (others => '0');
    else
        if rising_edge(clk) then
            MulGammaIn <= WaitAdd2_out;
            SAR2In     <= MulGammaOut;
            Add2I1     <= SAR2Out(15 downto 0);
            Add2I2     <= d3;
            d2s        <= Add2Out;
            d2stream   <= d2st;
        end if;
    end if;
end process;

-- Latching (Delta stage and sync after Delta stage):
process (clk, aclr) is
begin
    if aclr = '1' then
        MulDeltaIn    <= (others => '0');
        SAR3In        <= (others => '0');
        Add3I1        <= (others => '0');
        Add3I2        <= (others => '0');
```

```
        else
            if rising_edge(clk) then
                MulDeltaIn    <= WaitAdd3_out;
                SAR3In        <= MulDeltaOut;
                Add3I1        <= SAR3Out(15 downto 0);
                Add3I2        <= s3;
                s2stream      <= Add3Out;
            end if;
        end if;
    end process;

    sf <= s2stream;
    df <= d2final;

end architecture rtl;
```

Appendix B

Program Source Code

The source files for the test program are included below with the exception of the rangecoder and its quasi-static model. The source of these files can be found on the rangecoder website [12].

B.1 C-Source Files

B.1.1 ImComp.c

```
/*
   E.Kriegler
   22 August 2002
   An implementation of EZW Image Compression
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "dwt.h"
#include "ezw.h"
#include "unezw.h"
#include "ImComp.h"

#define versionStr "Date: 2003/04/11\n\n"

void loadRAW(int *image1, char *name, unsigned int size) {
   FILE *fp;
```

```
   unsigned char *buffer;
   unsigned int cnt;
   unsigned int x;
   unsigned int y;
   int errr;

   if ((buffer = malloc(size)) == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate file buffer\n\n");
      exit(1);
   }

   if ((fp = fopen(name, "rb")) == NULL) {
      printf("\n\nFATAL ERROR: Unable to open input image file\n\n");
      exit(1);
   }

   for (cnt = 0; cnt < size; cnt++) {
      if (fread(buffer, 1, size, fp) != size)   {
         if (feof(fp)) {
            printf("EOF ERROR");
         } else if ((errr = ferror(fp))) {
            printf("FERROR: %i", errr);
         }
         printf("\n\nFATAL ERROR: Failed to read line #%i
          of input file\n\n", cnt);
         exit(1);
      }
      y = cnt*size;
      for (x = 0; x < size; x++) {
         image1[x+y] = buffer[x];
      }
   }

   free(buffer);
   fclose(fp);
}
```

```
void saveRAW(int *image, char *name, unsigned int size) {
   FILE *fp;

   unsigned char *buffer;
   unsigned int cnt;
   unsigned int x;
   unsigned int y;

   if ((buffer = malloc(size)) == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate file buffer\n\n");
      exit(1);
   }

   if ((fp = fopen(name, "wb")) == NULL) {
      printf("\n\nFATAL ERROR: Unable to open input image file\n\n");
      exit(1);
   }

   for (cnt = 0; cnt < size; cnt++) {
      y = cnt*size;
      for (x = 0; x < size; x++) {
         buffer[x] = (unsigned char)image[x+y];
      }
      if (fwrite(buffer, 1, size, fp) != (size)) {
         printf("\n\nFATAL ERROR: Error writing output file\n\n");
         exit(1);
      }
   }

   free(buffer);
   fclose(fp);
}

void saveRAW16(int *image, char *name, unsigned int size) {
   FILE *fp;
```

```
   unsigned short int *buffer;
   unsigned int cnt;
   unsigned int x;
   unsigned int y;

   if ((buffer = malloc(size*2)) == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate file buffer\n\n");
      exit(1);
   }

   if ((fp = fopen(name, "wb")) == NULL) {
      printf("\n\nFATAL ERROR: Unable to open input image file\n\n");
      exit(1);
   }

   for (cnt = 0; cnt < size; cnt++) {
      y = cnt*size;
      for (x = 0; x < size; x++) {
         buffer[x] = (unsigned short int)image[x+y];
      }
      if (fwrite(buffer, 2, size, fp) != (size)) {
         printf("\n\nFATAL ERROR: Error writing output file\n\n");
         exit(1);
      }
   }

   free(buffer);
   fclose(fp);
}

int main() {

   char inName[12] = inFileName;
   char ezwName[12] = ezwFileName;
   char outName[12] = outFileName;
   int i;
```

```
int *band;

double MSE = 0;
double PSNR = 0;

int error;
double ME;
int maxError;

int levels;
int filter;
int budget;

int *origImage;
int *intImage;
int *workingMap1;

unsigned int t1;
float tf;

int *diffs;

struct ezw_stat *ezwStat;
struct unezw_stat *unezwStat;

int lctr, sctr;

FILE *ini;

printf("EZW Image Compression Test Software\n");
printf("E. Kriegler (Kriegler@sun.ac.za)\n");
printf("University of Stellenbosch\n");
printf(versionStr);

// ================================================
printf("Retrieving info from ini file:\n");
ini = fopen("ezw.ini", "r");
```

```
fgets(inName, 12, ini);
fscanf(ini, "%i %i %i", &levels, &filter, &budget);

printf(inName); printf("\n");
printf("levels: %i\n", levels);
printf("filter: %i\n", filter);
printf("budget: %i\n\n", budget);

fclose(ini);
// ================================================

if ((workingMap1 = malloc(SIZE*SIZE*sizeof(int))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate RAM for
     working Map\n\n");
   exit(1);
}
if ((origImage = malloc(SIZE*SIZE*sizeof(int))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate RAM for
     Image Backup\n\n");
   exit(1);
}
if ((intImage = malloc(SIZE*SIZE*sizeof(int))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate RAM for
     Integer Image\n\n");
   exit(1);
}
diffs = malloc(256*sizeof(int));

band = (int *)malloc(SIZE*SIZE*4);

printf("\nLoading Image:");
loadRAW(workingMap1, inName, SIZE);
printf(" done.\n");

for (i = 0; i < SIZE*SIZE; i++)   {
   origImage[i] = workingMap1[i];
```

```
}

// Wavelet Transform:
printf("\nPerforming Wavelet Decomposition: ");
fwt2d(workingMap1, SIZE, levels);
printf("done.\n");

for (i = 0; i < SIZE*SIZE; i++) {
   band[i] = abs(workingMap1[i]);
}
saveRAW16(band, "wavelet.raw", SIZE);
for (i = 0; i < SIZE*SIZE; i++) {
   band[i] = (abs(workingMap1[i])>>1);
   if (band[i] > 255) {
      band[i] = 255;
   }
}
saveRAW(band, "waveletl.raw", SIZE);

for (i = 0; i < SIZE*SIZE; i++) {
   intImage[i] = workingMap1[i];
}

// EZW Encoding:
printf("\nEZW Encoding Image:");
if ((ezwStat = malloc(sizeof(struct ezw_stat))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate memory for
     EZW Encoding\n\n");
   exit(1);
}
encode(workingMap1, SIZE, levels, ezwName, budget, ezwStat);
printf(" done.\n");

// EZW Decoding:
printf("\nEZW Decoding Image:");
if ((unezwStat = malloc(sizeof(struct unezw_stat))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate memory for
```

```
      EZW Decoding\n\n");
   exit(1);
}

for (i = 0; i < SIZE*SIZE; i++) {   // Reset workingMap to zeros
   intImage[i] = 0;
}
decode(intImage, SIZE, levels, ezwName, (*ezwStat).initThreshold,
   unezwStat);
printf(" done.\n");

for (i = 0; i < SIZE*SIZE; i++) {
   workingMap1[i] = intImage[i];
}

// Inverse Wavelet Transform:
printf("\nReconstructing Image: ");
iwt2d(workingMap1, SIZE, levels);
printf(" done.\n");

// Calculate Quality Factors:
printf("\nCalculating Quality Factor:");

maxError = 0;
ME = 0;
for (i = 0; i < 256; i++) {
   diffs[i] = 0;
}

lctr = 0; sctr = 0;

for (i = 0; i < SIZE*SIZE; i++)   {
   if (workingMap1[i] < 0) {
      sctr++;
      workingMap1[i] = 0;
   } else if (workingMap1[i] > 255) {
      lctr++;
```

```
            workingMap1[i] = 255;
        }

        t1 = abs(workingMap1[i] - origImage[i]);
        tf = (float)t1;
        MSE = MSE + (tf*tf);
        ME += tf;
        error = t1;

        if (error > maxError) {
            maxError = error;
        }
        diffs[error]++;
    }

    ME = ME / (SIZE*SIZE);
    MSE = MSE / (SIZE*SIZE);
    PSNR = 10*log((double)(255*255)/MSE)/log(10);
    printf(" done.\n");

    printf("\nSaving Image: ");
    for (i = 0; i < SIZE*SIZE; i++) {
        if ((workingMap1[i] > 255) && (workingMap1[i] < 0x80000000)) {
            printf("\n\n WARNING: LARGE POSITIVE NUMBERS ON IMAGE SAVE");
        }
        if (workingMap1[i] >= 0x80000000) {
            printf("\n\n WARNING: NEGATIVE NUMBERS ON IMAGE SAVE");
        }
    }
    saveRAW(workingMap1, outName, SIZE);
    printf("done.\n");

    //display stats:
    printf("\n\n EZW Statistics:\n");
    printf("\n Dominant Codes:    %8li", (*ezwStat).domCodes);
    printf("\n Subordinate Codes: %8li", (*ezwStat).subCodes);
    printf("\n Bits before Range: %8li", (*ezwStat).bitsMid);
```

```
   printf("\n Bytes Written:     %8i", (*ezwStat).bytesWritten);
   printf("\n Completed Passes:  %8i", (*ezwStat).passes);
   printf("\n Initial Threshold: %8i", (*ezwStat).initThreshold);
   printf("\n Final Threshold:   %8i", (*ezwStat).finThreshold);

   printf("\n\n UNEZW Statistics:\n");
   printf("\n Dominant Codes:    %8li", (*unezwStat).domCodes);
   printf("\n Subordinate Codes: %8li", (*unezwStat).subCodes);
   printf("\n Bits after Range:  %8li", (*unezwStat).bitsMid);
   printf("\n Completed Passes:  %8i", (*unezwStat).passes);
   printf("\n Final Threshold:   %8i", (*unezwStat).finThreshold);

   printf("\n\n Number of pixels >255:  %8i", lctr);
   printf("\n Number of pixels <0:    %8i", sctr);

   printf("\n\n QUALITY Statistics:\n");
   printf("\n MSE = %3.5f", MSE);
   printf("\n PSNR = %3.5fdB", PSNR);
   printf("\n Mean Absolute Error = %3.3f", ME);
   printf("\n Max Absolute Error = %i", maxError);


#ifdef breakdown

   printf("\n\n Breakdown of Absolute Error:\n");
   for (i = 0; i < _breakdownCnt; i++) {
      printf("\n Percentage of Errors between %i and %i pixel units:
        %3.2f%%", i, i+1, ((double)(diffs[i]*100)/(double)(SIZE*SIZE)));
   }

#endif

   printf("\n\n");

   //Free Allocated Memory
   free(workingMap1);
   free(origImage);
```

```
    free(intImage);
    free(ezwStat);
    free(unezwStat);
    free(diffs);
    free(band);

    return 0;
}
```

B.1.2 dwt.c

```
// Eduard Kriegler
// August 2002

// Software DWT Implementation

#include <stdio.h>
#include <stdlib.h>

#define alfaI   -1624
#define betaI     -54
#define gammaI     904
#define deltaI    454

#define kd0        832
#define kd1       1177
#define ks0       1260
#define ks1        891

void mI(int *vec, int N) {

    int i;
    int tmp;

    int start;
    int end;
    int half = N >> 1;
```

```
   start = half - 1;
   end = half;

   while (start > 0) {
      for (i = start; i < end; i = i + 2) {
         tmp = vec[i];
         vec[i] = vec[i+1];
         vec[i+1] = tmp;
      }
      start = start - 1;
      end = end + 1;
   }
}

void sI(int *vec, int N) {
   int i;
   int tmp;

   int start = 1;
   int end = N - 1;

   while (start < end) {
      for (i = start; i < end; i = i + 2) {
         tmp = vec[i];
         vec[i] = vec[i+1];
         vec[i+1] = tmp;
      }
      start = start + 1;
      end = end - 1;
   }
}

void fS1I(int *vec, int N, int factor) {

   int i;
   int half;
```

```
   int val;

   half = N >> 1;

   for (i = 0; i < half; i++) {
      if (i < half - 1) {
         val = (factor * (vec[i]+vec[i+1]) + 1) >> 10;
      } else if (i == (half - 1)) {
         val = (factor * (2*vec[i] - vec[i-1]) + 1) >> 10;
      }
      vec[half+i] = vec[half+i] + val;
   }
}

void fS2I(int *vec, int N, int factor) {

   int i;
   int half;

   int val;

   half = N >> 1;

   for (i = 0; i < half; i++) {
      if (i == 0) {
         val = (factor * (2*vec[half]) + 1) >> 10;
      } else if (i > 0) {
         val = (factor * (vec[half+i-1]+vec[half+i]) + 1) >> 10;
      }
      vec[i] = vec[i] + val;
   }
}

void iS1I(int *vec, int N, int factor) {

   int i;
```

```
   int half;

   int val;

   half = N >> 1;

   for (i = 0; i < half; i++) {
      if (i < half - 1) {
         val = (factor * (vec[i]+vec[i+1]) + 1) >> 10;
      } else if (i == (half - 1)) {
         val = (factor * (2*vec[i] - vec[i-1]) + 1) >> 10;
      }
      vec[half+i] = vec[half+i] - val;
   }
}

void iS2I(int *vec, int N, int factor) {

   int i;
   int half;

   int val;

   half = N >> 1;

   for (i = 0; i < half; i++) {
      if (i == 0) {
         val = (factor * (2*vec[half]) + 1) >> 10;
      } else if (i > 0) {
         val = (factor * (vec[half+i-1]+vec[half+i]) + 1) >> 10;
      }
      vec[i] = vec[i] - val;
   }
}

/*void fNI(int *vec, int N) {
   int i;
```

```
   int half = N >> 1;

   for (i = 0; i < half; i++) {
      vec[i] = (vec[i]*ks0) >> 9;
      vec[half+i] = (vec[half+i]*kd0) >> 9;
   }
}

void iNI(int *vec, int N) {
   int i;
   int half = N >> 1;

   for (i = 0; i < half; i++) {
      vec[i] = (vec[i]*ks1) >> 9;
      vec[half+i] = (vec[half+i]*kd1) >> 9;
   }
}*/

void fTI(int *vec, int N) {

   sI(vec, N);
   fS1I(vec, N, alfaI);
   fS2I(vec, N, betaI);
   fS1I(vec, N, gammaI);
   fS2I(vec, N, deltaI);
   //fNI(vec, N);
}

void iTI(int *vec, int N) {
   //iNI(vec, N);
   iS2I(vec, N, deltaI);
   iS1I(vec, N, gammaI);
   iS2I(vec, N, betaI);
   iS1I(vec, N, alfaI);
   mI(vec, N);
}
```

```
void fwt2d(int *map1, int size, int levels) {
   int currentSize = size;
   int level;

   int rowCtr;
   int colCtr;
   int i;

   unsigned int *invec1;
   unsigned int *source1;

   invec1 = malloc((currentSize) * sizeof(int));
   if (invec1 == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate temporary
        memory\n\n");
      exit(1);
   }

   source1 = map1;

   for (level = 1; level <= levels; level++) {

      for (rowCtr = 0; rowCtr < currentSize; rowCtr++) {

         fTI(source1, currentSize);
         source1 += size;

      }

      source1 = map1;

      for (colCtr = 0; colCtr < currentSize; colCtr++) {

         for (i = 0; i < currentSize; i++) {
            invec1[i] = source1[colCtr+(size*i)];
         }
         fTI(invec1, currentSize);
```

```
            for (i = 0; i < currentSize; i++) {
               source1[colCtr+(size*i)] = invec1[i];
            }

         }

         currentSize = currentSize >> 1;

      }

      free(invec1);
}

void iwt2d(int *map1, int size, int levels) {
      int currentSize = size;
      int level;

      int rowCtr;
      int colCtr;
      int i;

      unsigned int *invec1;
      unsigned int *source1;

      invec1 = malloc((currentSize) * sizeof(int));
      if (invec1 == NULL) {
         printf("\n\nFATAL ERROR: Unable to allocate temporary
           memory\n\n");
         exit(1);
      }
      source1 = map1;

      currentSize = size >> levels;

      for (level = 1; level <= levels; level++) {

         currentSize = currentSize << 1;
```

```
      for (colCtr = 0; colCtr < currentSize; colCtr++) {

         for (i = 0; i < currentSize; i++) {
            invec1[i] = source1[colCtr+(size*i)];
         }
         iTI(invec1, currentSize);
         for (i = 0; i < currentSize; i++) {
            source1[colCtr+(size*i)] = invec1[i];
         }

      }

      source1 = map1;

      for (rowCtr = 0; rowCtr < currentSize; rowCtr++) {
         iTI(source1, currentSize);
         source1 += size;
      }

      source1 = map1;

   }

   free(invec1);
}
```

B.1.3 ezw.c

```
/*
   E.Kriegler
   22 August 2002
   EZW Image Compression
*/

#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include "ezw.h"
#include "qsmodel.h"
#include "rangecod.h"
#include "ImComp.h"

#define zt_map

struct domelement {
   int x;
   int y;
   int code;
};

struct bandstruct {
   int size;
   int number;
   struct domelement *array;
};

struct bandstruct *band;
struct ezw_stat *estat;

qsmodel en_qsmDom4[4];
qsmodel en_qsmDom3[3];
qsmodel en_qsmSub[2];
rangecoder rc;
int lastSymD4 = 0;
int lastSymD3 = 0;
int lastSymS = 0;
int lastEncoder = 0;

int ZTStats = 0;
int ZTR_match = 0;

int *list;
long listCtr;
```

```
int imsize;
int nSubbands;
int *ztmap;
int *source;
FILE *fp;
long byteBudget;
int initThreshold;
int endEncoding;

int absMax(int *map, int size);
void domPass(int th);
void subPass(int th);
void addBit(int bit);
void addCode1(int code);
void addCode2(int code);
void processElement(int n, int i, int j, int th);
int zeroTree(int n, int i, int j, int th);
int zeroTree_map(int n, int i, int j, int th);
int zeroTree_recurse(int n, int i, int j, int th);
int zeroTree_match(int n, int i, int j, int th);
int zeroTree2(int n, int i, int j, int th);
void encodeD4(int sym);
void encodeD3(int sym);
void encodeS(int sym);
int lpt(int val, int th);
void genZTMap2(int size, int levels);
void genZTMap(int size, int levels);

// Output Data Handling:
int data = 0;
int cnt = 0;
unsigned char coded = 0;
int mask = 1;

int subSymbol = 0;
int subSymCtr = 0;
```

```
void encode(int *coef, int size, int levels, char *fname,
   long budget, struct ezw_stat *stat) {

   int i; int x; int y;
   int s = size >> levels;
   int sy;
   int th;

   int syfreq;
   int ltfreq;
   int sym;

   int freqs1[] = {d4_s0, d4_s1, d4_s2, d4_s3, d4_s4};
   int freqs2[] = {d3_s0, d3_s1, d3_s2, d3_s3};
   int freqs3[] = {s_s0, s_s1, s_s2};

   // General Parameters:
   nSubbands = (3*levels) + 1;
   source = coef;
   imsize = size;
   byteBudget = budget;
   endEncoding = 0;

   // Open the output file
   if ((fp = fopen(fname, "wb")) == NULL) {
      printf("\n\nFATAL ERROR: Unable to open EZW file for output\n\n");
      exit(1);
   }

   if ((ztmap = malloc(sizeof(int)*size*size)) == NULL) {
      printf("\n\nFATAL ERROR allocating memory for ZeroTree Map\n\n");
      exit(1);
   }
   // Create subordinate list
   if ((list = malloc(sizeof(int)*size*size)) == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate memory for
```

```
      Subordinate List\n\n");
   exit(1);
}
listCtr = 0;

// Create the subband array structure
if ((band = malloc(nSubbands*sizeof(struct bandstruct))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate memory for
     Dominant Pass\n\n");
   exit(1);
}

// Initialize subband array structure
for (i = 0; i < nSubbands; i++) {
   (band+i)->size = s;
   (band+i)->number = i;
   if (((band+i)->array = malloc(sizeof(struct domelement)*s*s))
      == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate memory for
       Dominant Pass\n\n");
      exit(1);
   }
   for (y = 0; y < s; y++) {
      sy = s * y;
      for (x = 0; x < s; x++) {
         (((band+i)->array)+x+sy)->code = NC;   // Not Coded

         if (i == 0) {
            (((band+i)->array)+x+sy)->x = x;
            (((band+i)->array)+x+sy)->y = y;
         } else {
            if (i % 3 == 1)
            {
               (((band+i)->array)+x+sy)->x = s+x;
               (((band+i)->array)+x+sy)->y = y;
            }
            if (i % 3 == 2)
```

```
            {
               (((band+i)->array)+x+sy)->x = x;
               (((band+i)->array)+x+sy)->y = s+y;
            }
            if (i % 3 == 0)
            {
               (((band+i)->array)+x+sy)->x = s+x;
               (((band+i)->array)+x+sy)->y = s+y;
            }
         }
      }
   }

   if ((i > 0) && (i % 3 == 0)) {
      s = s << 1;
   }
}

// Calculate Initial Threshold
initThreshold = 1 << (int)floor(log((double)absMax(source, imsize))
     /log(2));

// Generate the ZeroTree Map
genZTMap(size, levels);

// Setup stat structure
estat = stat;

(*estat).bitsMid = 0;
(*estat).bitsOut = 0;
(*estat).bytesWritten = 0;
(*estat).domCodes = 0;
(*estat).domTime = 0;
(*estat).finThreshold = 0;
(*estat).initThreshold = initThreshold;
(*estat).passes = 0;
(*estat).subCodes = 0;
```

```
(*estat).subTime = 0;
(*estat).totalTime = 0;

// Setup Entropy Encoder:
fpout = fp;
byteswritten = &(estat->bytesWritten);
for (sym = 0; sym < 4; sym++) {
   initqsmodel(en_qsmDom4+sym, 5, 12, dom4UpdFreq, freqs1, 1);
   if (sym != 3) {
      initqsmodel(en_qsmDom3+sym, 4, 12, dom3UpdFreq, freqs2, 1);
   }
   if ((sym != 2) && (sym != 3)) {
      initqsmodel(en_qsmSub+sym, 3, 12, subUpdFreq, freqs3, 1);
   }
}
start_encoding(&rc, 0, 0);

// Check for valid byte budget
if (byteBudget <= 0) {
   printf("\n\nFATAL ERROR: Illegal Byte Budget\n\n");
   exit(1);
}

// ##########################
// Begin Encoding Algorithm:
th = initThreshold;

while ((th != 0) && (endEncoding == 0)) {

   domPass(th);

   if (endEncoding == 0) {
      estat->passes++;
      subPass(th>>1);
   }
   if (endEncoding == 0) {
      estat->passes++;
```

```
            th = th >> 1;
            //sortSubList();
        }
    }

    #ifdef match
        printf("\nZTR Mismatches: %i\n", ZTR_match);
        printf("ZT Status Checks: %i\n\n", ZTStats);
    #endif

    estat->finThreshold = th;

    if (lastEncoder == 0) {
        qsgetfreq(en_qsmDom4+lastSymD4,4,&syfreq,&ltfreq);
    } else if (lastEncoder == 1) {
        qsgetfreq(en_qsmDom3+lastSymD3,3,&syfreq,&ltfreq);
    } else {
        qsgetfreq(en_qsmSub+lastSymS,2,&syfreq,&ltfreq);
    }
     encode_shift(&rc,syfreq,ltfreq,12);

    done_encoding(&rc);
    for (sym = 0; sym < 4; sym++) {
        deleteqsmodel(en_qsmDom4+sym);
        if (sym != 3) {
            deleteqsmodel(en_qsmDom3+sym);
        }
        if ((sym != 2) && (sym != 3)) {
            deleteqsmodel(en_qsmSub+sym);
        }
    }

    fclose(fp);                 // Close output file
    free(list);                 // Deallocate subordinate list

    // Deallocate subband array structure
    for (i = 0; i < nSubbands; i++) {
```

```
      free((band+i)->array);
   }
   free(band);
   free(ztmap);

}

void domPass(int th) {
   int n;
   int i;
   int j;
   int size;
   int sizej;

   int nodeStatus; int parentNodeStatus;
   for (n = 0; n < nSubbands; n++) {
      size = (band+n)->size;
      for (j = 0; j < size; j++) {
         sizej = size*j;
         for (i = 0; i < size; i++) {

            // Check Node Status: (Has it been coded before?)
            nodeStatus = (((band+n)->array)+i+sizej)->code;

            if ((nodeStatus != POS) && (nodeStatus != NEG)) {
               // Check Parent Node Status:
            // (Must not code if part of ZEROTREE)
               if (n < 4) { // Special case for coarsest scales
                  if (n == 0) {
                     // No Parent
                // Anything except IZ and ZTR
                     parentNodeStatus = POS;
                  } else {
                     // Parent has same coordinates in subband 0
                     parentNodeStatus = ((band->array)+i+sizej)->code;
                  }
               } else {
```

```
            // Parent node coordinates is current coors >> 1
            // in subband n-3.
            parentNodeStatus = (((band+n-3)->array)+(i>>1)+
           (size>>1)*(j>>1))->code;
         }
         if (parentNodeStatus == ZTR) {
            (((band+n)->array)+i+sizej)->code = ZTR;
         } else {
            processElement(n, i, j, th);
            if (n < nSubbands-3) {
               switch ((((band+n)->array)+i+sizej)->code) {
                  case ZTR: {
                     addCode1(ZTR);
                     estat->domCodes++;
                     break;
                  }
                  case IZ: {
                     addCode1(IZ);
                     estat->domCodes++;
                     break;
                  }
                  case POS: {
                     addCode1(POS);
                     estat->domCodes++;
                     break;
                  }
                  case NEG: {
                     addCode1(NEG);
                     estat->domCodes++;
                     break;
                  }
               }
            } else {
               switch ((((band+n)->array)+i+sizej)->code) {
                  case ZTR: {
                     addCode2(ZTR);
                     estat->domCodes++;
```

```
                        break;
                    }
                    case POS: {
                        addCode2(POS);
                        estat->domCodes++;
                        break;
                    }
                    case NEG: {
                        addCode2(NEG);
                        estat->domCodes++;
                        break;
                    }
                }
            }

            // Check byte budget
            if (estat->bytesWritten >= byteBudget) {
                endEncoding = 1;
                break;
            }
        }
    }
    if (endEncoding == 1) {
        break;
    }
  }
  if (endEncoding == 1) {
    break;
  }
 }
}

void subPass(int th) {
  int i;

  subSymbol = 0;
```

```
   subSymCtr = 0;

   if ((th > 0) && (listCtr > 0)) {
      for (i = 0; i < listCtr; i++) {
         // Check byte budget:
         if (estat->bytesWritten >= byteBudget) {
            endEncoding = 1;
            break;
         }
         if ((*(list+i) & th) != 0) {
            // Encode 1
            addBit(1);
            estat->subCodes++;
         } else {
            // Encode 0
            addBit(0);
            estat->subCodes++;
         }
      }
   }

}

void processElement(int n, int i, int j, int th) {
   //struct domelement *map;
   int x; int y;
   int size = (*(band+n)).size;

   int sizej   = size*j;
   int imsizey;

   x = (((band+n)->array)+i+sizej)->x;
   y = (((band+n)->array)+i+sizej)->y;

   imsizey = imsize*y;

   if (abs((*(source+x+imsizey))) >= th) {    // Check SIGNIFICANCE
```

```
      if ((*(source+x+imsizey)) >= 0) {       // POSITIVE Significant
         (((band+n)->array)+i+sizej)->code = POS;
         *(list+listCtr) = *(source+x+imsizey);
         listCtr++;
         (*(source+x+imsizey)) = 0;
      } else {                              // NEGATIVE Significant
         (((band+n)->array)+i+sizej)->code = NEG;
         *(list+listCtr) = -*(source+x+imsizey);
         listCtr++;
         (*(source+x+imsizey)) = 0;
      }
   } else {                                 // ZERO TREE Status
      if (zeroTree(n, i, j, th) == 1) {
         (((band+n)->array)+i+sizej)->code = ZTR;
      } else {
         (((band+n)->array)+i+sizej)->code = IZ;
      }
   }
}

int zeroTree(int n, int i, int j, int th) {
   // Return 1 if coefficient is a ZeroTree root
   // or 0 if not a ZTR (in which case it is a IZ)

   ZTStats++;

   #ifdef zt_map
      return zeroTree_map(n, i, j, th);
   #endif

}

int zeroTree_map(int n, int i, int j, int th) {

   int x; int y;
   int size = (*(band+n)).size;
```

```
   x = (((band+n)->array)+i+size*j)->x;
   y = (((band+n)->array)+i+size*j)->y;

   if (n >= (nSubbands - 3)) {
     // If in 3 finest scales, insignificant coef is a ZTR
      return 1;
   }
   if ((ztmap[x+y*(imsize)] & th) == 0) {
      return 1;
   } else {
      return 0;
   }
}


void addBit(int bit) {
   estat->bitsMid++;
   encodeS(bit);
}

void addCode1(int code) {
   int outCode;

   switch (code) {
      case ZTR: {
         outCode = 0;
         break;
      }
      case IZ: {
         outCode = 1;
         break;
      }
      case POS: {
         outCode = 2;
         break;
      }
      case NEG: {
```

```
            outCode = 3;
            break;
        }
        default: {
            outCode = 0;
        }
    }
    encodeD4(outCode);
    estat->bitsMid += 2;
}

void addCode2(int code) {
    int outCode;

    switch (code) {
        case ZTR: {
            outCode = 0;
            break;
        }
        case POS: {
            outCode = 1;
            break;
        }
        case NEG: {
            outCode = 2;
            break;
        }
        default: {
            outCode = 0;
        }
    }
    encodeD3(outCode);
    estat->bitsMid += 2;
}

int absMax(int *map, int size) {
    int amax = 0;
```

```
   int i;

   for (i = 0; i < size*size; i++) {
      if (abs(*(map+i)) > amax) {
         amax = abs(*(map+i));
      }
   }
   return amax;
}

void encodeD4(int sym) {
   int syfreq;
   int ltfreq;

   if ((sym < 0) || (sym > 3)) {
      printf("\n\nFATAL ERROR: Symbol to encode outside
        bounds: %i\n\n", sym);
      exit(1);
   }

   qsgetfreq(en_qsmDom4+lastSymD4, sym, &syfreq, &ltfreq);
   encode_shift(&rc, syfreq, ltfreq, 12);
   qsupdate(en_qsmDom4+lastSymD4, sym);
   lastSymD4 = sym;
   lastEncoder = 0;
}

void encodeD3(int sym) {
   int syfreq;
   int ltfreq;

   if ((sym < 0) || (sym > 2)) {
      printf("\n\nFATAL ERROR: Symbol to encode outside
        bounds: %i\n\n", sym);
      exit(1);
   }
```

```
   qsgetfreq(en_qsmDom3+lastSymD3, sym, &syfreq, &ltfreq);
   encode_shift(&rc, syfreq, ltfreq, 12);
   qsupdate(en_qsmDom3+lastSymD3, sym);
   lastSymD3 = sym;
   lastEncoder = 1;
}

void encodeS(int sym) {
   int syfreq;
   int ltfreq;

   if ((sym != 0) && (sym != 1)) {
      printf("\n\nFATAL ERROR: Symbol to encode outside
        bounds: %i\n\n", sym);
      exit(1);
   }

   qsgetfreq(en_qsmSub+lastSymS, sym, &syfreq, &ltfreq);
   encode_shift(&rc, syfreq, ltfreq, 12);
   qsupdate(en_qsmSub+lastSymS, sym);
   lastSymS = sym;
   lastEncoder = 2;
}

// Returns largest power of 2 smaller than 'val'
int lpt(int val, int th) {
   int i = th;

   while ((i > 0) && (i > abs(val))) {
      i = i >> 1;
   }

   return i;
}

void genZTMap(int size, int levels) {
```

```
int x;
int y;
int i;

int ztmvp;
int ztmvc1;
int ztmvc2;
int ztmvc3;
int ztmvc4;

int ztmsize = size >> 1;
int ztmsiz2 = size >> 2;

int sizey;
int sizey2;

int t;
double v;

for (y = 0; y < size; y++) {
   sizey = size*y;
   for (x = 0; x < size; x++) {
      v = source[x+sizey];
      t = lpt((int)v, initThreshold);
      ztmap[x+sizey] = t;
   }
}

i = levels - 1;

while (i > 0) {
   for (y = ztmsiz2; y < ztmsize; y++)   {
      sizey = size*y;
      sizey2 = size*(y<<1);
      for (x = 0; x < ztmsiz2; x++) {
         ztmvp = ztmap[x+sizey];
         ztmvc1 = ztmap[(x<<1)+sizey2];
```

```
            ztmvc2 = ztmap[(x<<1)+1+sizey2];
            ztmvc3 = ztmap[(x<<1)+size+sizey2];
            ztmvc4 = ztmap[(x<<1)+1+size+sizey2];
            ztmap[x+sizey] = ztmvp | ztmvc1 | ztmvc2 | ztmvc3 | ztmvc4;
         }
      }
      for (y = ztmsiz2; y < ztmsize; y++)   {
         sizey = size*y;
         sizey2 = size*(y<<1);
         for (x = ztmsiz2; x < ztmsize; x++) {
            ztmvp = ztmap[x+sizey];
            ztmvc1 = ztmap[(x<<1)+sizey2];
            ztmvc2 = ztmap[(x<<1)+1+sizey2];
            ztmvc3 = ztmap[(x<<1)+size+sizey2];
            ztmvc4 = ztmap[(x<<1)+1+size+sizey2];
            ztmap[x+sizey] = ztmvp | ztmvc1 | ztmvc2 | ztmvc3 | ztmvc4;
         }
      }
      for (y = 0; y < ztmsiz2; y++)   {
         sizey = size*y;
         sizey2 = size*(y<<1);
         for (x = ztmsiz2; x < ztmsize; x++) {
            ztmvp = ztmap[x+sizey];
            ztmvc1 = ztmap[(x<<1)+sizey2];
            ztmvc2 = ztmap[(x<<1)+1+sizey2];
            ztmvc3 = ztmap[(x<<1)+size+sizey2];
            ztmvc4 = ztmap[(x<<1)+1+size+sizey2];
            ztmap[x+sizey] = ztmvp | ztmvc1 | ztmvc2 | ztmvc3 | ztmvc4;
         }
      }
      ztmsize = ztmsize >> 1;
      ztmsiz2 = ztmsiz2 >> 1;
      i--;
   }
   for (y = 0; y < ztmsize; y++) {
      sizey = size*y;
      for (x = 0; x < ztmsize; x++) {
```

```
            ztmvp  = ztmap[x+sizey];
            ztmvc1 = ztmap[x+ztmsize+sizey];
            ztmvc2 = ztmap[x+size*(y+ztmsize)];
            ztmvc3 = ztmap[x+ztmsize+size*(y+ztmsize)];
            ztmap[x+sizey] = ztmvp | ztmvc1 | ztmvc2 | ztmvc3;
         }
      }

}
```

B.1.4 unezw.c

```
/*
   E.Kriegler
   22 August 2002
   EZW Image Decompression
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "unezw.h"
#include "qsmodel.h"
#include "rangecod.h"
#include "ImComp.h"

#define NC      0
#define POS      1
#define NEG      2
#define IZ      3
#define ZTR      4
#define END      99999

struct domelement {
   int x;
   int y;
   int code;
```

```
};

struct bandstruct {
   int size;
   int number;
   struct domelement *array;
};

struct bandstruct *band;
struct unezw_stat *estat;

qsmodel de_qsmDom4[4];
qsmodel de_qsmDom3[3];
qsmodel de_qsmSub[2];
rangecoder drc;
int dlastSymD4 = 0;
int dlastSymD3 = 0;
int dlastSymS = 0;
int lastDecoder = 0;

int *listx;
int *listy;
long listCtr;

int imsize;
int nSubbands;
int *dest;
int *destUpper;
int *destLower;
FILE *fp;
//FILE *debug2;
int initThreshold;
int endOfFile;

// Decoding Variables:
int ddata = 0;
int dcoded = 0;
```

```
int dcnt = 0;
int dmask = 0x100;
int dsubSymbol = 0;
int dsubSymCtr = 0;

// Function Prototypes
void ddomPass(int th);
void dsubPass(int th);
int getByte();
int HMdecode();
int getDomSymbol(int select);
int getSubSymbol();
void _printMap();

void decode(int *coef, int size, int levels, char *fname,
   long initTh, struct unezw_stat *stat) {
   int s = size >> levels;
   int x; int y; int i;
   int th;
   int sym;

   int freqs1[] = {d4_s0, d4_s1, d4_s2, d4_s3, d4_s4};
   int freqs2[] = {d3_s0, d3_s1, d3_s2, d3_s3};
   int freqs3[] = {s_s0, s_s1, s_s2};

   //int syfreq;
   //int ltfreq;

   // General Parameters:
   imsize = size;   // Image size (side length)
   dest = coef;   // Image is created here

   nSubbands = (3*levels) + 1;      // Number of Subbands in image
   initThreshold = initTh;         // Starting Threshold
   estat = stat;               // Statistics Structure
   endOfFile = 0;
```

```
// Setup Subordinate List
if ((listx = malloc(size*size*sizeof(int))) == NULL) {
   printf("\n\nFATAL ERRORS: Unable to allocate memory for
     SubList\n\n");
   exit(1);
}
if ((listy = malloc(size*size*sizeof(int))) == NULL) {
   printf("\n\nFATAL ERRORS: Unable to allocate memory for
     SubList\n\n");
   exit(1);
}
listCtr = 0;

// Open the input file
if ((fp = fopen(fname, "rb")) == NULL) {
   printf("\n\nFATAL ERROR: Unable to open EZW file for input\n\n");
   exit(1);
}



// Create the subband array structure
if ((band = malloc(nSubbands*sizeof(struct bandstruct))) == NULL) {
   printf("\n\nFATAL ERROR: Unable to allocate memory for Subband
    Array Structure1\n\n");
   exit(1);
}

// Initialize subband array structure
for (i = 0; i < nSubbands; i++) {
   (band+i)->size = s;
   (band+i)->number = i;
   if (((band+i)->array = malloc(sizeof(struct domelement)*s*s))
     == NULL) {
      printf("\n\nFATAL ERROR: Unable to allocate memory for Subband
      Array Structure2\n\n");
      exit(1);
   }
```

```
      for (y = 0; y < s; y++) {
         for (x = 0; x < s; x++) {
            (((band+i)->array)+x+s*y)->code = NC;   // Not Coded

            if (i == 0) {
               (((band+i)->array)+x+s*y)->x = x;
               (((band+i)->array)+x+s*y)->y = y;
            } else {
               if (i % 3 == 1)
               {
                  (((band+i)->array)+x+s*y)->x = s+x;
                  (((band+i)->array)+x+s*y)->y = y;
               }
               if (i % 3 == 2)
               {
                  (((band+i)->array)+x+s*y)->x = x;
                  (((band+i)->array)+x+s*y)->y = s+y;
               }
               if (i % 3 == 0)
               {
                  (((band+i)->array)+x+s*y)->x = s+x;
                  (((band+i)->array)+x+s*y)->y = s+y;
               }
            }
         }
      }

      if ((i > 0) && (i % 3 == 0)) {
         s = s << 1;
      }
   }

   // Setup stat structure
   estat = stat;

   (*estat).bitsIn = 0;
   (*estat).bitsMid = 0;
```

```
(*estat).domCodes = 0;
(*estat).domTime = 0;
(*estat).finThreshold = 0;
(*estat).passes = 0;
(*estat).subCodes = 0;
(*estat).subTime = 0;
(*estat).totalTime = 0;

// Setup Entropy Encoder:
fpout = fp;
for (sym = 0; sym < 4; sym++) {
   initqsmodel(de_qsmDom4+sym, 5, 12, dom4UpdFreq, freqs1, 0);
   if (sym != 3) {
      initqsmodel(de_qsmDom3+sym, 4, 12, dom3UpdFreq, freqs2, 0);
   }
   if ((sym != 2) && (sym != 3)) {
      initqsmodel(de_qsmSub+sym, 3, 12, subUpdFreq, freqs3, 0);
   }
}
start_decoding(&drc);


// Setup decode maps:
destUpper = malloc(imsize*imsize*sizeof(int));
destLower = malloc(imsize*imsize*sizeof(int));

for (i = 0; i < imsize*imsize; i++) {
   destUpper[i] = 0;
   destLower[i] = 0;
}

// ######################
// Begin Decoding Loop:

th = initThreshold;

while ((th > 0) && (endOfFile == 0)) {
```

```
      ddomPass(th);

      if (endOfFile == 0) {
         estat->passes++;
         dsubPass(th >> 1);
      }
      if (endOfFile == 0) {
         estat->passes++;
         th = th >> 1;
      }

   }

   estat->finThreshold = th;

   for (i = 0; i < imsize*imsize; i++) {
      dest[i] = (destUpper[i]+destLower[i]) >> 1;
   }

    done_decoding(&drc);
   for (sym = 0; sym < 4; sym++) {
      deleteqsmodel(de_qsmDom4+sym);
      if (sym != 3) {
         deleteqsmodel(de_qsmDom3+sym);
      }
      if ((sym != 2) && (sym != 3)) {
         deleteqsmodel(de_qsmSub+sym);
      }
   }

   fclose(fp);                  // Close input file
   free(listx);               // Deallocate subordinate list
   free(listy);               // Deallocate subordinate list

   free(destUpper);
   free(destLower);
```

```
   // Deallocate subband array structure
   for (i = 0; i < nSubbands; i++) {
      free((band+i)->array);
   }
   free(band);
}

void _printMap() {
}


void ddomPass(int th) {
   int n; int i; int j;
   int nodeStatus; int parentNodeStatus = 0;
   int symbol;
   int x; int y;

   n = 0;

   while ((n < nSubbands) && (endOfFile == 0)) {
      i = 0; j = 0;
      while ((j < (band+n)->size) && (endOfFile == 0)) {
         i = 0;
         while ((i < (band+n)->size) && (endOfFile == 0)) {

            nodeStatus = (((band+n)->array)+i+((band+n)->size*j))->code;

            if ((nodeStatus != POS) && (nodeStatus != NEG)) {
               if (n < 4) {
                  if (n == 0) {
                     parentNodeStatus = POS;
                  } else {
                     parentNodeStatus = ((band->array)+i+
                      (band->size*j))->code;
                  }
               } else {
```

```
      parentNodeStatus = (((band+n-3)->array)+(i>>1)+
    ((band+n-3)->size*(j>>1)))->code;
  }
  if (parentNodeStatus == ZTR) {
      (((band+n)->array)+i+((band+n)->size*j))->code = ZTR;
  } else {
      if (n < nSubbands-3) {
          symbol = getDomSymbol(0);
      } else {
          symbol = getDomSymbol(1);
      }
      if (symbol == END) {
          endOfFile = 1;
          break;
      }
      estat->domCodes++;
      (((band+n)->array)+i+((band+n)->size*j))->code =
    symbol;
      switch (symbol) {
          case POS: {
              x = (((band+n)->array)+i+((band+n)->size*j))->x;
              y = (((band+n)->array)+i+((band+n)->size*j))->y;
              destLower[x+imsize*y] = th;
              destUpper[x+imsize*y] = (2*th)-1;
              *(listx+listCtr) = x;
              *(listy+listCtr) = y;
              listCtr++;
              break;
          }
          case NEG: {
              x = (((band+n)->array)+i+((band+n)->size*j))->x;
              y = (((band+n)->array)+i+((band+n)->size*j))->y;
              destLower[x+imsize*y] = -th;
              destUpper[x+imsize*y] = -((2*th)-1);
              *(listx+listCtr) = x;
              *(listy+listCtr) = y;
              listCtr++;
```

```
                        break;
                    }
                    case IZ: {
                        break;
                    }
                    case ZTR: {
                        break;
                    }
                }
            }
        }
        i++;
    }
    j++;
  }
  n++;
 }
}

void dsubPass(int th) {
  int i = 0;
  int bit;
  int x; int y;

  dsubSymbol = 0;
  dsubSymCtr = 0;

  if ((th > 0) && (listCtr > 0)) {
    while ((i < listCtr) && (endOfFile == 0)) {
      x = listx[i]; y = listy[i];
      bit = getSubSymbol();
      if (bit == END) {
        endOfFile = 1;
        break;
      }

      estat->subCodes++;
```

```
            if (bit == 1) {
               destLower[x+imsize*y] += (destUpper[x+imsize*y] -
               destLower[x+imsize*y]) / 2;
            } else {
               destUpper[x+imsize*y] -= (destUpper[x+imsize*y] -
               destLower[x+imsize*y]) / 2;
            }
            i++;
         }
      }

}

int readD4() {
   int code;
   int ltfreq;
   int syfreq;

    ltfreq = decode_culshift(&drc,12);
    code = qsgetsym(de_qsmDom4+dlastSymD4, ltfreq);
    if (code == 4) { /* check for end-of-file */
      return END;
   }
    qsgetfreq(de_qsmDom4+dlastSymD4,code,&syfreq,&ltfreq);
    decode_update( &drc, syfreq, ltfreq, 1<<12);
    qsupdate(de_qsmDom4+dlastSymD4,code);
   dlastSymD4 = code;

   lastDecoder = 0;

   return code;
}

int readD3() {
   int code;
   int ltfreq;
   int syfreq;
```

```
    ltfreq = decode_culshift(&drc,12);
    code = qsgetsym(de_qsmDom3+dlastSymD3, ltfreq);
    if (code == 3) { /* check for end-of-file */
      return END;
   }
    qsgetfreq(de_qsmDom3+dlastSymD3,code,&syfreq,&ltfreq);
    decode_update( &drc, syfreq, ltfreq, 1<<12);
    qsupdate(de_qsmDom3+dlastSymD3,code);
   dlastSymD3 = code;

   lastDecoder = 1;

   return code;
}

int readS() {
   int code;
   int ltfreq;
   int syfreq;

    ltfreq = decode_culshift(&drc,12);
    code = qsgetsym(de_qsmSub+dlastSymS, ltfreq);
    if (code == 2) { /* check for end-of-file */
      return END;
   }
    qsgetfreq(de_qsmSub+dlastSymS,code,&syfreq,&ltfreq);
    decode_update( &drc, syfreq, ltfreq, 1<<12);
    qsupdate(de_qsmSub+dlastSymS,code);
   dlastSymS = code;

   lastDecoder = 2;

   return code;
}

int getDomSymbol(int select) {
```

```
int decoded;

if (select == 0) {      // work with 4-symbol alphabet
   while (1 == 1) {
      decoded = readD4();
      switch (decoded) {
         case 0: {
            return ZTR;
         }
         case 1: {
            return IZ;
         }
         case 2: {
            return POS;
         }
         case 3: {
            return NEG;
         }
         case END: {
            return END;
         }
      }
   }
} else {            // work with 3-symbol alphabet
   while (1 == 1) {
      decoded = readD3();
      switch (decoded) {
         case 0: {
            return ZTR;
         }
         case 1: {
            return POS;
         }
         case 2: {
            return NEG;
         }
         case END: {
```

```
                return END;
            }
        }
    }
}
}

int getSubSymbol() {
   return readS();
}
```

B.2 Header Files

B.2.1 ImComp.h

```
// Eduard Kriegler
// August 2002

#define SIZE 1024   // Image Size

// add this line to see histogram of absolute errors
#define breakdown

#define ezwFileName "output.ezw"   // EZW output filename
#define outFileName "recon.raw"      // Output Image

#define _absDiffMult 16
#define _breakdownCnt 24

// Codes used for EZW coding - DO NOT CHANGE
#define NC      0
#define POS      1
#define NEG      2
#define IZ      3
#define ZTR      4

// Range Coder Defines:
```

```
#define dom4UpdFreq 35   // Update rate for dompass 4+1 symmodel
#define dom3UpdFreq 280   // Update rate for dompass 3+1 symmodel
#define subUpdFreq 32   // Update rate for subpass model
// 35 280 32

// dominant pass 4+1 symbol model freq init:
#define d4_s0   2840
#define d4_s1   398
#define d4_s2   426
#define d4_s3   432
#define d4_s4   0

// dominant pass 3+1 symbol model freq init: (For finest scales)
#define d3_s0   3734
#define d3_s1   177
#define d3_s2   185
#define d3_s3   0

// subordinate pass 2+1 symbol model freq init:
#define s_s0   2721
#define s_s1   1375
#define s_s2   0
```

B.2.2 dwt.h

```
// Eduard Kriegler
// August 2002

void fwt2d(int *map1, int size, int levels);
void iwt2d(int *map1, int size, int levels);
```

B.2.3 ezw.h

```
// Eduard Kriegler
// August 2002

struct ezw_stat {
   long totalTime;      // Total execution time
```

```
   long domTime;      // Time spent in dominant passes
   long subTime;      // Time spent in subordinate passes

   long domCodes;      // Number of dominant codes outputted
   long subCodes;      // Number of subordinate codes outputted

   long bitsMid;      // Number of bits outputted (before range)
   long bitsOut;      // Final number of bits outputted

   int bytesWritten;   // Number of bytes written to disk

   int passes;         // Number of completed passes
   int initThreshold;   // Initial (starting) threshold
   int finThreshold;   // Final (ending) threshold
};

void encode(int *coef, int size, int levels, char *fname,
   long byteBudget, struct ezw_stat *stat);
```

B.2.4 unezw.h

```
// Eduard Kriegler
// August 2002

struct unezw_stat {
   long totalTime;      // Total execution time
   long domTime;      // Time spent in dominant passes
   long subTime;      // Time spent in subordinate passes

   long domCodes;      // Number of dominant codes read
   long subCodes;      // Number of subordinate codes read

   long bitsIn;      // Number of bits read from disc
   long bitsMid;      // Number of bits after entropy decoding
```

```
    int passes;          // Number of completed passes
    int finThreshold;    // Final (ending) threshold
};

void decode(int *coef, int size, int levels, char *fname,
        long initTh, struct unezw_stat *stat);
```

Appendix C

Profiling and Assembly Implementation Information

Section C.1 explain how the profiling information was obtained and Section C.2 contains the results.

C.1 Obtaining Profiling and Assembly Implementation Information

The examples in this section use the GNU C-Compiler (GCC), the GNU Profiling tool (gprof) and the GNU Coverage tool (gcov). The process described below utilises the GNU Tools running under Cygwin, but the process should be very similar for the GNU Tools running on Linux distributions.

To obtain profiling information, the program source code must be compiled with special parameters:

```
> gcc -O0 -pg -fprofile-arcs -ftest-coverage -o imcomp.exe imcomp.c ezw.c
   dwt.c rangecod.c qsmodel.c
```

It is important to note that the optimisation level is explicitly set to zero ("-O0"). As is explained later on, coverage data in addition to profiling data is required. Any level of optimisation can render coverage data useless because of instruction reordering and simplifications.

The next parameter is "-pg". This tells the compiler and linker to add extra code to generate the profile information.

The "-fprofile-arcs" parameters tell the compiler to add count that will count the number of times each arc is executed. It is required for both profiling and coverage tests.

Coverage tests also require the "-ftest-coverage" parameter. With this parameter the compiler will generate data files for the GNU Coverage tool.

The final parameter sets the output executable file and names the input source files.

The next step is to run the compiled program. Running the program will generate the wanted data. After the program execution ended, the profiling tool can be run:

```
> gprof imcomp.exe > imcomp.prof.txt
```

The profiling tool only requires the name of the executable that must be profiled. The pipe-redirection is used to store the output in a text file for later reference instead of having it all dumped on the console screen.

Finally each C-source file is tested for coverage. This is necessary because many functions contains branches (like if-statements and switch-statements) and also loops (for-loops, while-loops, etc). The number of times each branch or loop is executed will affect the WCET of the function. Running the coverage test will generate this information.

It is necessary to run the gcov program on each source file separately. The source files cannot be grouped on the command line.

```
> gcov -b -c -f ezw.c
```

Parameter "-b" informs the program to calculate how often each branch in the program was taken. The second parameter, "-c", selects that the branch execution be shown as a number of executions rather than a percentage.

To get a summary of the coverage information of each function in the source file, the "-f" parameter is used.

The next step in calculating the WCET of each function is to obtain the assembly implementation of each function.

Obtaining the assembly implementation requires the installation of the GCC Tool-chain for the target processor. In this case the MIPS compiler was used. The MIPS GCC tool-chain is available for download from the MIPS website [10].

```
> sde-gcc -S -mmad -O0 ezw.c
```

By using the "-S" parameter in invoking gcc, the compiler will stop the compilation process after the C-source has been compiled to assembler instructions. The assembler instructions can now be found in text files with the same filename as the original C-source files, but with the ".s" extension. Again the optimisation level is explicitly set to zero. This will generate the worst case code.

The "-mmad" parameter is used to tell the compiler to use the MUL instruction instead of MULT which is slower.

C.2 Profiling and Assembly Analysis Results

The results of the analysis are displayed in Tables C.2 and C.1. The blocks listed in the tables refer to the code inside if-statements, for-loops and while-loops.

Function/ Subblock	Instructions				
	Single Cycle	Multiply	Divide	Load/Store	Branch
absMax	4	0	0	7	1
absMax - block 1	14	1	0	15	4
addBit	5	0	0	11	2
addCode1	9	0	0	17	7
addCode2	9	0	0	17	7
deleteqsmodel	4	0	0	13	5
domPass	4	0	0	6	1
domPass - block 1a	12	0	0	6	1
domPass - block 1b	20	0	0	25	9
domPass - block 2	33	1	0	23	7
domPass - block 3	4	1	0	9	5
domPass - block 4	8	0	0	10	5
done_encoding	37	0	0	45	19
dorescale	16	0	2	71	14
dorescale - block 1	11	0	0	26	3
dorescale - block 2	4	0	0	9	3
dorescale - block 3	8	0	0	17	2
enc_normalise	4	0	0	5	1
enc_normalise - block 1	3	0	0	11	4
enc_normalise - block 2	12	0	0	19	5
encode_shift	10	2	0	28	3
encodeD3	24	0	0	14	4
encodeD4	24	0	0	14	4
encodeS	24	0	0	14	4
genZTMap	7	0	0	12	1
genZTMap - block 1a	2	1	0	6	2
genZTMap - block 1b	9	0	0	19	4
genZTMap - block 2a	4	2	0	9	2
genZTMap - block 2b	33	0	0	42	3
genZTMap - block 3a	4	2	0	9	2

Table C.1: Continued on next page

Function/ Subblock	Instructions				
	Single Cycle	Multiply	Divide	Load/Store	Branch
genZTMap - block 3b	33	0	0	42	3
genZTMap - block 4a	4	2	0	9	2
genZTMap - block 4b	33	0	0	42	3
genZTMap - block 5	5	0	0	14	3
genZTMap - block 6a	2	1	0	6	2
genZTMap - block 6b	27	2	0	40	3
initqsmodel	17	0	0	40	7
lpt	4	0	0	7	1
lpt - block 1	5	0	0	6	4
processElement	40	2	0	38	2
processElement - block 1a	24	0	0	23	2
processElement - block 1b	14	0	0	10	3
qsgetfreq	10	0	0	20	1
qsupdate	10	0	0	24	3
resetqsmodel	7	0	0	15	3
resetqsmodel - block 1	6	0	0	12	3
start_encoding	6	0	0	17	1
subPass	4	0	0	10	3
subPass - block 1	8	0	0	16	6
zeroTree_Map	42	3	0	34	4

Table C.1: Assembly Instruction Counts

Function/ Subblock	Source File	Worst Case Execution Count
absMax	ezw	1
absMax - block 1		262144
addBit	ezw	61784
addCode1	ezw	120343
addCode2	ezw	162082
deleteqsmodel	qsmodel	9
domPass	ezw	10
domPass - block 1a		2324918
domPass - block 1b		268844

Table C.2: Continued on next page

Function/ Subblock	Source File	Worst Case Execution Count
domPass - block 2		2613615
domPass - block 3		15250
domPass - block 4		220
done_encoding	rangecod	1
dorescale	qsmodel	9809
dorescale - block 1		16667
dorescale - block 2		0
dorescale - block 3		0
enc_normalise	rangecod	288700
enc_normalise - block 1		96
enc_normalise - block 2		32002
encode	ezw	1
encode_shift	rangecod	288699
encodeD3	ezw	162082
encodeD4	ezw	120343
encodeS	ezw	61784
genZTMap	ezw	1
genZTMap - block 1a		512
genZTMap - block 1b		262144
genZTMap - block 2a		252
genZTMap - block 2b		21840
genZTMap - block 3a		252
genZTMap - block 3b		21840
genZTMap - block 4a		252
genZTMap - block 4b		21840
genZTMap - block 5		6
genZTMap - block 6a		4
genZTMap - block 6b		16
initqsmodel	qsmodel	9
lpt	ezw	262144
lpt - block 1		2429031
processElement	ezw	268844
processElement - block 1a		115082
processElement - block 1b		175062

Table C.2: Continued on next page

Function/ Subblock	Source File	Worst Case Execution Count
qsgetfreq	qsmodel	288699
qsupdate	qsmodel	288698
resetqsmodel	qsmodel	9
resetqsmodel - block 1		38
start_encoding	rangecod	1
subPass	ezw	9
subPass - block 1		61784
zeroTree_Map	ezw	175062

Table C.2: Worst Case Execution Counts

Figure C.1 contains a bar-graph of the WCET of each individual function. Clearly most of the time is spent in the dominant pass function and this function should be the first target for optimisation.

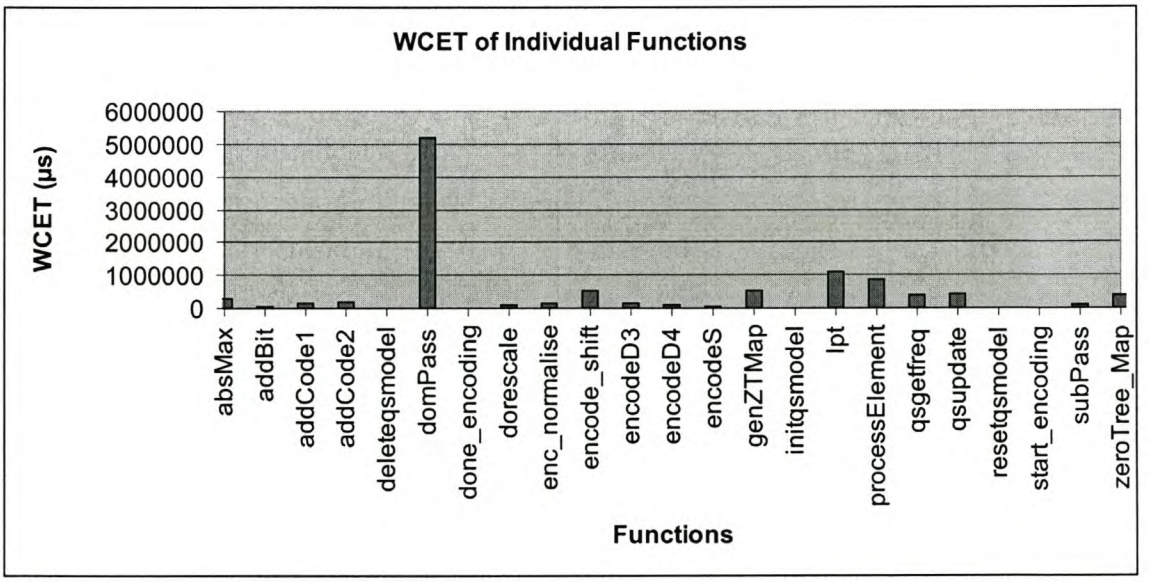

Figure C.1: *WCET of the Individual Functions*

Appendix D

Images Used in Testing the EZW Implementation

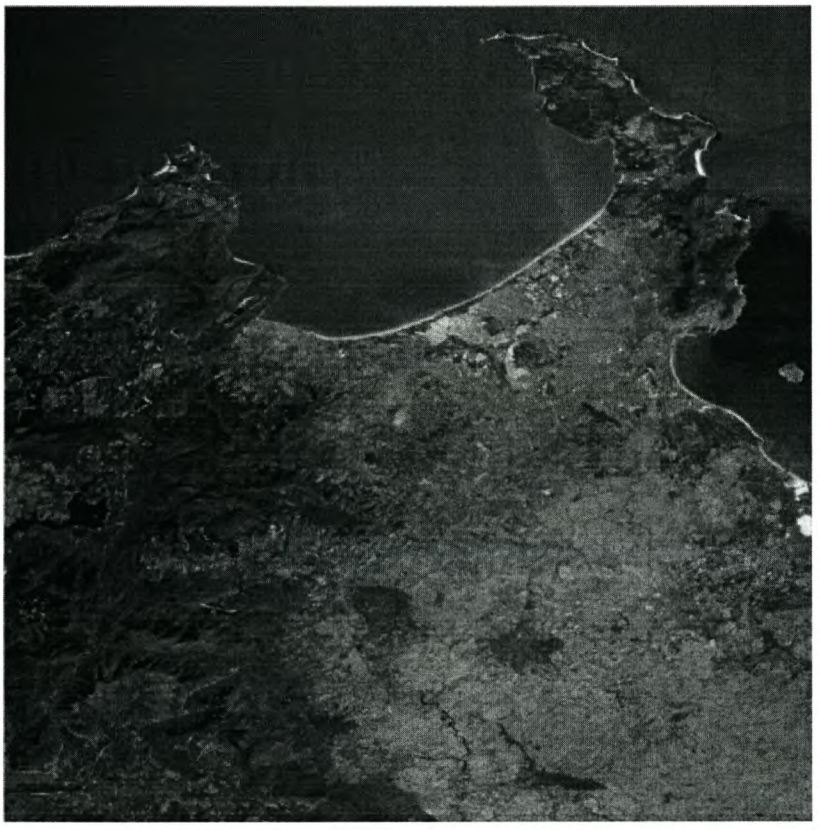

Figure D.1: *SatCape Image*

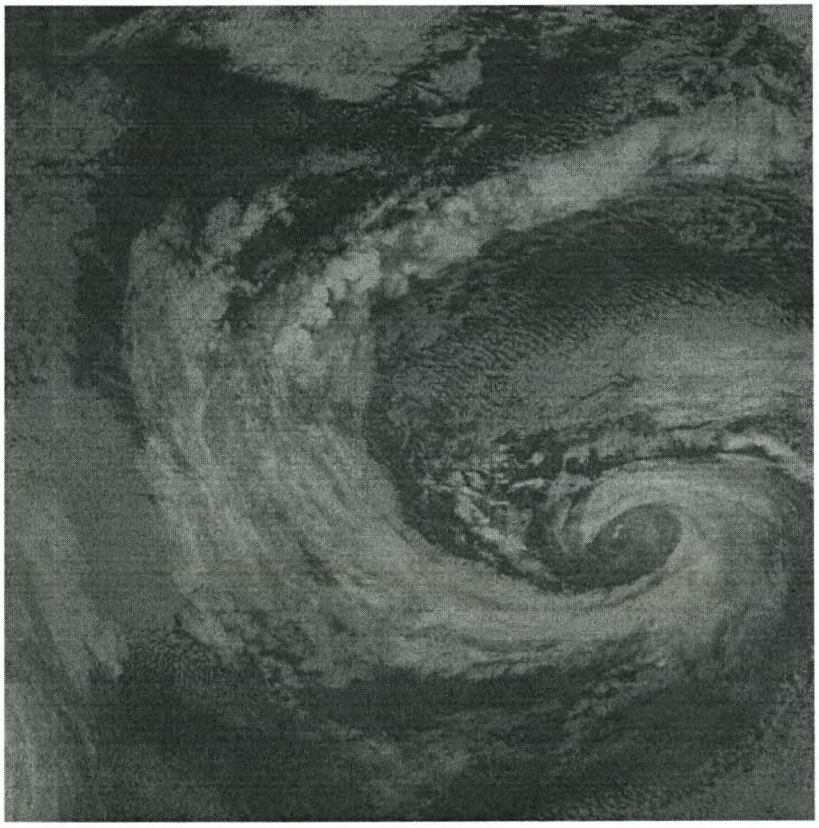

Figure D.2: *North Atlantic Image*

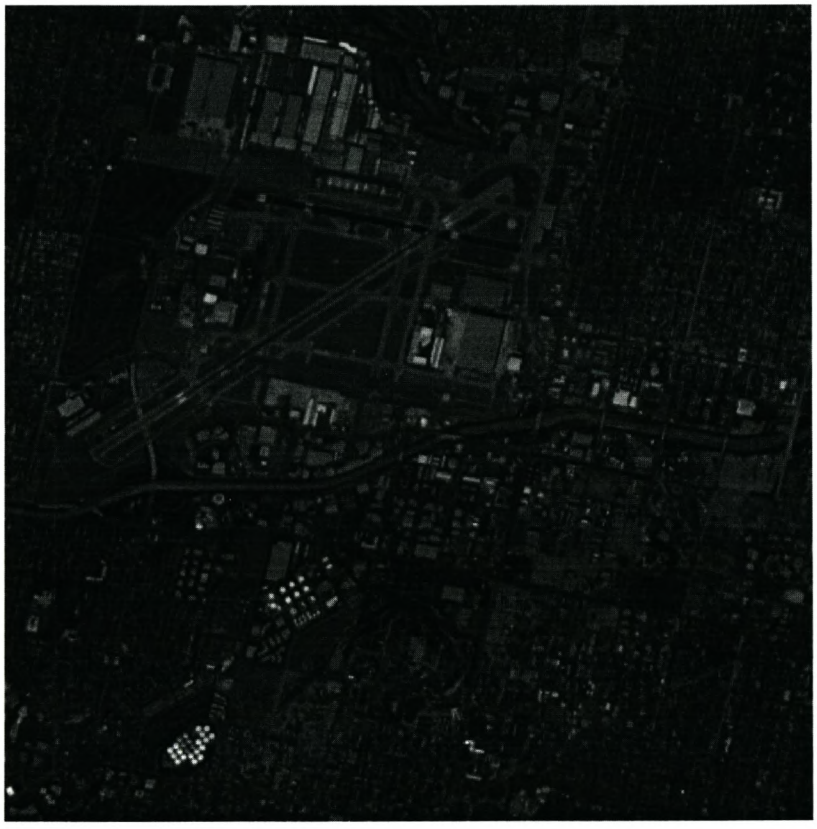

Figure D.3: *Spot_LA_Panchr Image*

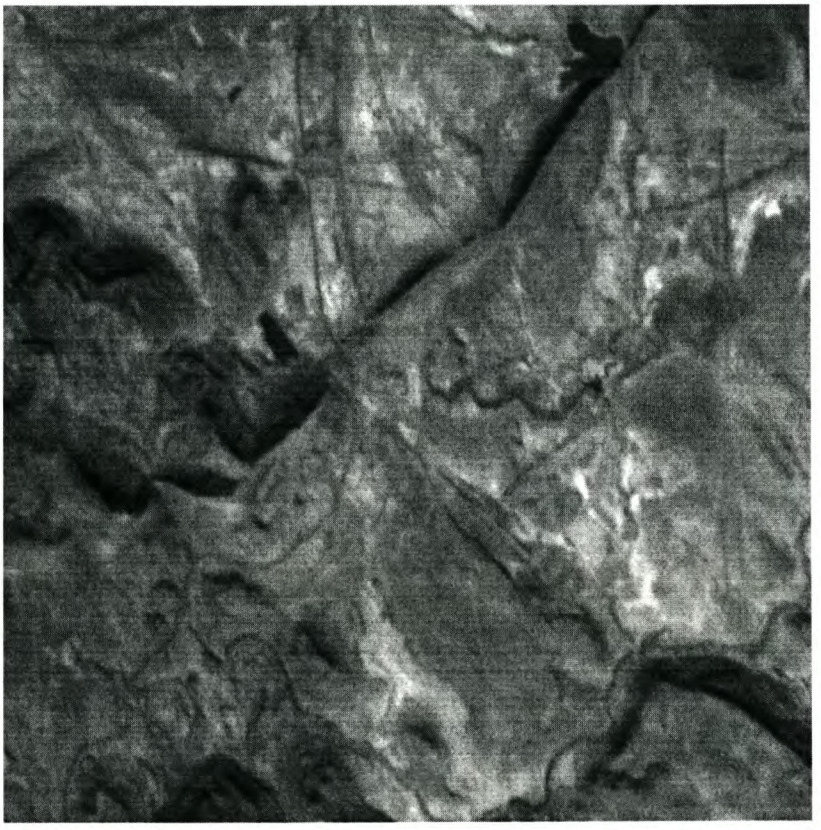

Figure D.4: *Transkei Image*

Figure D.5: *Lena Image*